THE INTERPLAY OF REWARD AND ENERGY IN REAL-TIME SYSTEMS

by

Cosmin Rusu

B.S, Technical University of Cluj-Napoca, Romania, 2000

Submitted to the Graduate Faculty of

Arts and Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Cosmin Rusu

It was defended on

August 8, 2006

and approved by

Dr. Rami Melhem

Dr. Daniel Mossé

Dr. Bruce R. Childers

Dr. Prashant Krishnamurthy

Dissertation Advisors: Dr. Rami Melhem,

Dr. Daniel Mossé

Copyright © by Cosmin Rusu
2006

ABSTRACT

THE INTERPLAY OF REWARD AND ENERGY IN REAL-TIME SYSTEMS

Cosmin Rusu, PhD

University of Pittsburgh, 2006

This work contends that three constraints need to be addressed in the context of power-aware real-time systems: energy, time and task rewards/values. These issues are studied for two types of systems. First, embedded systems running applications that will include temporal requirements (e.g., audio and video). Second, servers and server clusters that have timing constraints and Quality of Service (QoS) requirements implied by the application being executed (e.g., signal processing, audio/video streams, webpages). Furthermore, many future real-time systems will rely on different software versions to achieve a variety of QoS-aware tradeoffs, each with different rewards, time and energy requirements.

For hard real-time systems, solutions are proposed that maximize the system reward/profit without exceeding the deadlines and without depleting the energy budget (in portable systems the energy budget is determined by the battery charge, while in server farms it is dependent on the server architecture and heat/cooling constraints). Both continuous and discrete reward and power models are studied, and the reward/energy analysis is extended with multiple task versions, optional/mandatory tasks and long-term reward maximization policies.

For soft real-time systems, the reward model is relaxed into a QoS constraint, and stochastic schemes are first presented for power management of systems with unpredictable workloads. Then, load distribution and power management policies are addressed in the context of servers and homogeneous server farms. Finally, the work is extended with QoS-aware local and global policies for the general case of heterogeneous systems.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS 1
1.0 INTRODUCTION 2
2.0 BACKGROUND AND RELATED WORK 7
2.1 REAL-TIME SYSTEMS 7
2.2 REWARD-BASED COMPUTING 8
2.3 ENERGY-AWARE COMPUTING 10
2.4 DYNAMIC VOLTAGE SCALING 11
2.5 POWER MANAGEMENT IN SERVER FARMS 13
2.6 WORKLOAD ESTIMATION 14
3.0 SYSTEM MODELS AND PROBLEM DESCRIPTION 16
3.1 REAL-TIME APPLICATION MODELS 16
3.1.1 Frame-based and Periodic Task Sets 16
3.1.2 Aperiodic Request Servers 17
3.2 REWARD MODELS 17
3.2.1 Continuous Reward Functions 17
3.2.2 Discrete Rewards and Task Versions 18
3.2.3 Quality of Service for Aperiodic Requests 19
3.3 POWER MODELS 20
3.3.1 Continuous Speeds/Voltages 21
3.3.2 Discrete Speed/Voltage Levels 21
3.3.3 System Power Model 22
3.4 PROBLEM DESCRIPTION AND RESEARCH OVERVIEW 23

3.5 EVALUATION METHODOLOGY 25
4.0 POWER CONSUMPTION ESTIMATION 27
4.1 FINE-GRAINED CPU POWER MODEL 28
4.1.1 The PPC405 Architecture 28
4.1.2 Power Modeling Events 29
4.1.3 Power Modeling Validation 31
4.2 COARSE-GRAINED SYSTEM POWER MODEL 33
4.3 CHAPTER SUMMARY 37
5.0 ENERGY-AWARE REWARD MAXIMIZATION FOR HARD REAL-TIME SYSTEMS 39
5.1 CONTINUOUS REWARDS AND POWER FUNCTIONS 39
5.1.1 Problem Definition 40
5.1.2 Properties of the Optimal Solution 41
5.1.3 Optimal Solutions for Specific Power Functions 46
5.1.3.1 Optimal solution for identical power functions 47
5.1.3.2 Optimal solution for certain polynomial power functions . . . 47
5.1.4 Iterative DVS Algorithm for General Power Functions 48
5.1.4.1 Inter-Task Transfers 48
5.1.4.2 Intra-Task Transfers 49
5.1.4.3 Reduction 49
5.1.4.4 Algorithm 49
5.1.5 Evaluation 51
5.2 DISCRETE REWARDS AND POWER FUNCTIONS 53
5.2.1 Optional Single Version 54
5.2.1.1 The REW-Pack Algorithm 55
5.2.1.2 The REW-Unpack Algorithm 58
5.2.1.3 Evaluation 58
5.2.2 Multiple Task Versions 60
5.2.2.1 The MV-Pack Algorithm 61
5.2.2.2 Evaluation 63

5.3 LONG-TERM REWARD MAXIMIZATION 65
5.3.1 Battery-Powered Systems 65
5.3.2 Rechargeable Systems 66
5.3.2.1 Rechargeability Background 66
5.3.2.2 Models and Problem Definition 68
5.3.2.3 Static Energy Allocation 70
5.3.2.4 Dynamic Energy Allocation Schemes 72
5.3.2.5 Experimental Results 74
5.4 CHAPTER SUMMARY 76
6.0 QOS-AWARE ENERGY MINIMIZATION FOR SOFT REAL-TIME SYSTEMS 79
6.1 STOCHASTIC DVS FOR UNPREDICTABLE WORKLOADS 80
6.1.1 Two Request-Driven Signal-Processing Applications 80
6.1.2 System Model 82
6.1.3 Prediction Schemes 83
6.1.3.1 Application-Oblivious Prediction (AO) 83
6.1.3.2 Application-Aware Prediction (AA) 84
6.1.4 A Stochastic DVS Algorithm 85
6.1.5 Experimental Results 88
6.2 HOMOGENEOUS REAL-TIME CLUSTERS 92
6.2.1 Cluster Model 92
6.2.2 Load-Aware On/Off DVS (LAOVS) 93
6.2.2.1 Load Estimation 93
6.2.2.2 Threshold Schemes 94
6.2.2.3 Workload Distribution 95
6.2.2.4 Experiments on a Testbed 95
6.3 HETEROGENEOUS REAL-TIME CLUSTERS 97
6.3.1 Cluster Model 98
6.3.2 Global Power Management 99
6.3.2.1 Load Definition and Estimation 99

6.3.2.2 Server Information 100
6.3.2.3 On/Off Policy 101
6.3.2.4 Request Distribution Policy 104
6.3.2.5 Implementation Issues 105
6.3.3 Local Power Management 106
6.3.3.1 Local DVS Policy 107
6.3.3.2 Implementation Issues 107
6.3.4 Evaluation 109
6.3.4.1 DVS Policy 109
6.3.4.2 Overall Scheme 110
6.4 CHAPTER SUMMARY 111
7.0 CONCLUSIONS 114
BIBLIOGRAPHY 119

LIST OF TABLES

4.1 Power modeling events 31
4.2 Energy validation results 32
4.3 Parameters of the machines of the Apache cluster 34
4.4 Idle/busy power consumption (in Watts) for each Apache server 34
4.5 Web server statistics: static page sizes and dynamic page running times . . . 36
5.1 Frequency/voltage settings for Intel XScale processors 58
6.1 Request execution times (in millions of cycles) 81
6.2 Request inter-arrival times (in seconds) 81
6.3 PPC405LP power consumption of SBT and CAF applications 82
6.4 Evaluation of DVS policies for unpredictable workloads 90
6.5 IBM PowerPC 750 frequencies and system total power (measured) 95
6.6 System energy consumption for RR and LAOVS, normalized to RR/NPM . . 97
6.7 Energy management for servers and server clusters 98

LIST OF FIGURES

3.1 Typical reward functions . . . 18
3.2 Research overview . . . 23
4.1 The PPC405GP pipeline . . . 28
4.2 Hardware versus simulated power graphs . . . 33
4.3 Power consumption versus load for Apache servers . . . 37
5.1 Design space . . . 43
5.2 Transfer types . . . 48
5.3 Fixed-size transfers, identical power functions . . . 51
5.4 Variable-size transfers, polynomial power functions . . . 53
5.5 Variable-size transfers, realistic power functions . . . 54
5.6 Flowchart of *REW-Pack* . . . 56
5.7 Evaluation of *REW-Pack* and *REW-Unpack* . . . 60
5.8 Flowchart of *MV-Pack* . . . 62
5.9 Evaluation of *MV-Pack* . . . 64
5.10 Current-voltage characteristic of a typical solar cell . . . 67
5.11 Charging and discharging characteristics of rechargeable batteries . . . 68
5.12 Solar power patterns . . . 69
5.13 Static versus dynamic energy allocation policies . . . 75
5.14 Chapter overview: reward maximization for hard real-time systems with energy constraints . . . 76
6.1 Power and energy consumption for the SBT 81 minutes trace . . . 90
6.2 Comparison of the DVS policies using synthetic traces . . . 91

6.3 Cluster architecture 92

6.4 The power consumption for PowerPC 750 systems (RR vs LAOVS) 97

6.5 Online cluster power and energy estimation 106

6.6 Evaluation of local and global power management techniques 110

6.7 Chapter overview: QoS-aware energy minimization for soft real-time systems 111

ACKNOWLEDGEMENTS

First of all, I would like to thank my two advisors, Rami Melhem and Daniel Mossé, for their permanent guidance, and steady confidence in my research. This thesis would have not been possible without your efforts and enthusiasm. I very much appreciated your constant availability for discussion, and your support through some more difficult times. I am also grateful to my committee members, Bruce Childers and Prashant Krishnamurthy, for their insightful comments on my work and thesis.

A big thanks goes to my colleagues in the Power-Aware Real-Time Systems (PARTS) group. I really enjoyed your friendship, and our lively discussions and debates in a very pleasant working environment. Many thanks to Nevine AbouGhazaleh for six years of sharing the same office, Ruibin Xu and Dakai Zhu for our collaboration on several papers, Alexandre Ferreira for his friendship and affability (not to mention fixing any hardware problem we run into), and our outside lab visitors Claudio Scordino and Luciano Bertini for becoming so quickly my friends. It was a privilege working with you all.

Finally, I would like to thank my family and my many friends outside school that made life in Pittsburgh enjoyable. And, since I earned my right to write anything here, many thanks to the Steelers and "The Bus" for winning the Super Bowl in what could be my last year in this wonderful city.

1.0 INTRODUCTION

Expanding beyond the traditional focus on performance, power management is now a well established area in computer systems research. This is specially true in the embedded arena, but also in servers; it comes from the desire to extend device lifetime (e.g., flying across the Pacific in a single battery) and/or cost of energy (e.g., paying less than 15% of total costs for energy in data centers). In these arenas, the trend is a paradigm shift from a maximum performance approach to a power/performance tradeoff.

However, reducing power and energy does not come for free and is not suitable for all systems. Some systems are amenable to such tradeoffs, specially systems where desired properties are well specified. Real-time systems research provides a framework for specifying such requirements, specially the timing requirements of an application. Some domains are hard real-time (such as mission critical applications, where deadline misses result in system failure), while other domains are soft real-time, with statistically specified performance requirements. Note that low-power and high-performance are typically conflicting goals. For example, one of the well-known solutions to the power consumption problem is to run the devices at slower speeds (less performance). Thus, real-time issues should be analyzed in conjunction with power management issues, so that the effect one has on the other can be controlled.

In addition to power and performance, a third dimension that should be considered is system reward (also referred to as utility, value or QoS). The utility of an application relates to the accuracy of the produced results, which in turn depends on the amount of resources (such as CPU time) allotted to the application. Ideally, systems monitor themselves and adapt the resource allocation to maximize application utilities.

This dissertation is dedicated to studying the interplay of system reward and power

management in the context of real-time systems. Rather than explicitly trading reward for energy, one of the dimensions (i.e., reward or energy) is fixed and given as a system requirement, while the other dimension (energy and reward respectively) represents the optimization objective. Thus, two types of problems are analyzed: maximizing the system reward for real-time systems with a fixed energy budget, and minimizing the energy consumption for real-time systems with a fixed reward (QoS) requirement. This corresponds to two types of systems, embedded systems with fixed energy budgets and high-performance server systems with fixed QoS requirements respectively.

In the embedded arena, most mobile, wireless, systems-on-a-chip and other computing-in-the-small devices have energy constraints, embodied by a battery that has a finite lifetime. Therefore, an essential element of these embedded systems is the way in which power is managed. In addition to the power management needs, some of these devices execute real-time applications, in which producing timely results is typically as important as producing logically correct outputs.

In the high-performance servers arena, power consumption is becoming a major concern in large server farms. Electricity cost is already a significant fraction of the operation cost of data centers, with that fraction likely to increase in the future [64, 8]. Furthermore, clusters with high peak power need complex and expensive cooling infrastructures to ensure the proper operation of the servers. With power densities increasing due to increasing performance and QoS demands and tighter packing, manufacturers are facing the problem of building powerful systems without introducing additional cooling techniques such as liquid cooling.

The reward/energy analysis can be particularly beneficial in real-time systems where admission control algorithms are traditionally used to only accept tasks that will finish before their deadlines. The main problem with admission control algorithms is that they are conservative, and that they under-utilize resources. For example, typical real-time tasks only take 10% to 40% of their worst case execution time (WCET) [28].

A better alternative to admission control is to allow systems to run above the load restrictions imposed by real-time constraints. These overloaded systems lend themselves naturally to scenarios in which some applications are executed in lieu of more important applications

or a different, perhaps less accurate, version of an application is executed. Application areas in which an approximate but timely result is acceptable include multimedia [65], and image and speech processing [18, 31, 86]. In these applications a reward function can be used to assign a value to the application, as a function of the amount of computation resources alloted to the application. The problem is then how to choose the amount of resources to allot to applications that will maximize the overall reward given to the system, such that the real-time and energy constraints of the system are still satisfied.

Version programming opens a new opportunity for QoS tradeoffs. An example of version programming comes from satellite-based signal processing [84]. In that example, four different algorithms (least-mean square, maximum-likelihood, software trigger, matched filter) with running times ranging from microseconds to hundreds of milliseconds, and energy consumptions from μJoules to Joules provide different levels of accuracy. Another example is Automatic Target Recognition (ATR), where task values and execution times are roughly proportional to the number of targets detected [37]. Note that, in general, task versions result not only from different algorithms, but also from the same application with different input arguments (such as encoding/decoding at different rates, low/high quality compression schemes, low/high resolution image processing), or even different invocation periods [14].

The three constraints mentioned, namely energy, performance (deadline), and reward (QoS) play important roles in the current generation of embedded devices. An optimal scheme chooses to run the most valued applications or versions of applications, without depleting the energy source while still meeting all deadlines. In high-end systems (such as server farms) request arrival rates are unpredictable and the reward is typically relaxed into a QoS specification (such as meeting 95% of deadlines without dropping any requests). In such systems, an optimal scheme minimizes the energy consumption while guaranteeing the QoS.

In summary, this dissertation studies the interplay between energy and reward in the context of real-time systems. Specifically, the contributions of this work are as follows:

- Extending previous work that only considers two of the constraints at a time, a system reward maximization algorithm is developed for continuous power and reward models, that is the first to simultaneously consider energy, deadline and rewards. In our exper-

iments, the proposed iterative algorithm is shown to be within 1% (on average) of the optimal [69].

- A new discrete reward model is proposed for applications that do not reward partial execution, as for example, tasks with different versions. Three DVS (dynamic voltage scaling) algorithms are proposed for discrete rewards and the realistic case of discrete power functions. In our experimental setting, the algorithms were found to have average errors within 3% of the optimal with running times in the microsecond range [68, 70].
- Long-term reward maximization schemes are studied for systems with energy constraints. While the previous algorithms are used for short-term reward maximization, energy allocation policies are analyzed for long-term (lifetime) reward maximization of battery-powered embedded systems. Both static and dynamic energy allocation schemes are devised. Three dynamic schemes are shown to outperform the static allocation by using various energy reclamation mechanisms [71, 72].
- An efficient stochastic dynamic voltage scaling (DVS) scheme is developed for soft real-time systems with unpredictable workloads, and is shown to largely outperform prediction-based and utilization-based schemes [73].
- Local (DVS) and global (on/off) schemes are combined in the context of soft real time homogeneous server clusters. The approach is based on evaluating the system load and is shown to outperform previous frequency-based schemes. The scheme was implemented in a prototype embedded cluster [92, 93].
- Local and global QoS-aware power management algorithms are proposed for the general case of heterogeneous server clusters. The proposed solution is based on offline power measurements and is the first work that combines global and local power management for heterogeneous clusters. A utilization-based DVS scheme is proposed for local power management, and heterogeneous power-aware request distribution policies are devised for global power management. The scheme was implemented and evaluated on a real Apache web server cluster [67].

In addition, to better understand the energy consumption of applications, a fine-grained event-based power model for PowerPC405 embedded processors (PPC405GP and PPC405LP)

is devised and shown to have an average error within 6% when compared to real measurements of embedded benchmarks [77, 78]. Similarly, a coarse-grained measurement-based approach is found to estimate the energy consumption of an entire cluster within 1% of actual measurements for the web traces used in our experimental setting [67].

The organization of this dissertation is as follows. In Chapter 2, the basic definitions are given and the current research status is examined in the area of real-time systems, energy aware computing and reward-based computing. The system models, problem descriptions and research overview are presented in Chapter 3. Chapter 4 evaluates power/energy estimation techniques for both embedded and high-performance systems. Chapters 5 and 6 address the interplay of energy and reward for hard and soft real-time systems respectively. Finally, we draw our conclusions in Chapter 7.

2.0 BACKGROUND AND RELATED WORK

2.1 REAL-TIME SYSTEMS

Real-time systems research provides a framework for power/performance tradeoffs by specifying the timing requirements of applications. In general, real-time applications start executing after their *ready time* (the time when the application becomes available for execution) and finish their execution before their *deadline*. For *hard real-time* systems the deadline is a strict requirement and deadline misses may result in system failure. Other domains are *soft real-time*, meaning that occasional deadline misses are tolerable and the performance requirement is a statistical one.

Scheduling in real-time systems means determining *which* task is executed at *what* time. For multiprocessor systems scheduling also determines *where* (that is, on which processing unit) the task is to be executed. In addition, tasks may be either *preemptive* or *non-preemptive*. A schedule is said to be *feasible* if all constraints (timing and resource constraints) of all tasks are satisfied. A scheduling algorithm can be either *fixed priority* or *dynamic priority* (when task priorities change over time).

For hard real-time systems, typical analysis is performed considering *worst-case execution times (WCET)* because scheduling of such systems must be guaranteed even in worst-case scenarios. If a new task cannot be accommodated in an existing schedule, an *admission control* algorithm may decide to reject the task. Unfortunately WCET analysis is pessimistic [28] and typically results in severe under-utilization of the system. WCET analysis, however, is the only option for guaranteeing the feasibility of critical hard real-time systems.

As the hard real-time constraint is relaxed, a better option is to allow systems to execute above the restrictions imposed by admission control algorithms, and dynamically exploit the

slack created by early task completions. The analysis becomes then a statistical one, such as maintaining the deadline miss ratio below a certain percentage. Application adaptation (discussed in detail in the next section) refers to increasing or reducing the amount of resources allotted to applications, and provides great flexibility for such overloaded systems.

A plethora of scheduling algorithms exists for various real-time systems models, both single and multi-processor. Most common hard real-time models are *periodic* (with deadlines equal to periods) and *frame-based* (where all tasks have equal deadlines and periods). For soft real-time systems a variety of *rate-based* models are in use (such as the *(m,k)* model that requires only m out of every k task invocations meet their deadlines [38]) as well as a general *aperiodic* model where tasks can arrive randomly, each with its own specified deadline. For single processors, the seminal work of Liu and Layland has established the optimality of *earliest deadline first (EDF)* for dynamic priority and *rate monotonic scheduling (RMS)* for fixed priority systems [51]. For ideal systems (no overheads) *generalized processor sharing (GPS)* was proposed [60], with several extensions for parallel systems (such as *proportional fairness* or *P-fair* [7]).

Scheduling algorithms with focus on system reward or power management for single processor systems are further reviewed in the next sections.

2.2 REWARD-BASED COMPUTING

The *IC (Imprecise Computation)* [53, 81] and *IRIS (Increased Reward with Increased Service)* [23, 46] models were proposed to enhance the resource utilization and provide graceful degradation in real-time systems. In the IC model every real-time task is composed of a mandatory part and an optional part. The mandatory part must be completed before the task's deadline to yield an output of minimal quality. The optional part is to be executed after the mandatory part while still before the deadline. The longer the optional part executes, the better the quality of the result (reward). The algorithms proposed for IC applications concentrate on a model that has an upper bound on the execution time and reward that can be assigned to the optional part and the aim is usually to minimize the (weighted) sum of errors. Several efficient algorithms have been proposed to optimally solve the schedul-

ing problem of aperiodic tasks [53, 81]. A common assumption in these studies is that the quality of the results produced is a linear function of the precision error; more general error functions are not usually addressed.

An alternative model is the IRIS model with no upper bounds on the execution times of the tasks and no separation between the mandatory and optional parts (i.e., tasks may be allotted no CPU time). Typically, a non-decreasing concave reward function is associated with each task's execution time. In [23, 24] the problem of maximizing the total reward in a system of aperiodic tasks was addressed and an optimal solution for static task sets was presented, as well as two extensions that include mandatory parts and policies for dynamic task arrivals.

Both IC and IRIS focus on linear and concave (logarithmic for example) functions because they represent most of the real-world applications, such as image and speech processing [18, 31, 86], multimedia applications [65], information gathering [35] and database query processing [87]. The case of real applications with no reward for partial executions or using step functions has been shown in [53] to be NP-Complete. Furthermore, the reward-based scheduling problem for convex reward functions is NP-Hard, as shown in [5]. Periodic reward-based scheduling for *error-cumulative* (errors have an effect on future instances of the same task) and *error non-cumulative* applications was explored in [19]. An optimal algorithm assuming concave reward functions and error non-cumulative applications was presented in [5].

In [65] a QoS-based resource allocation model (Q-RAM) was proposed for periodic applications, with reward functions in terms of utilization of resources. In the Q-RAM framework each application requires a minimum resource allocation to perform acceptably, and can improve its utility with larger resource allocations. Q-RAM targets systems in which multiple applications with various requirements along multiple QoS dimensions are competing for resources. An iterative algorithm was first presented for the case of one resource and multiple QoS dimensions [65]. In [66], the Q-RAM work is continued by the authors with the solution for a particular audio-conferencing application with two resources (CPU utilization and network bandwidth) and one QoS dimension (sampling rate). Several resource tradeoffs (e.g., compression schemes to reduce network bandwidth while increasing the number of

CPU cycles) are also investigated, assuming linear utility functions.

A similar utility-based approach for resource allocation in wireless networks is proposed in [22]. The resource under consideration is network bandwidth. QoS levels of wireless connections are specified using discrete resource-utility functions, enhanced with various connection factors (such as age of the connection and disconnection penalty).

Recently, *time utility functions (TUF)* were also proposed for describing utility for aperiodic tasks. A TUF is a generalization of the deadline constraint, assigning the reward of an applications as a function of the application completion time [50].

Step reward functions remain generally unstudied. Note as well that prior to this dissertation, the three constraints (namely energy, deadlines and rewards) were treated separately.

2.3 ENERGY-AWARE COMPUTING

Energy management is nowadays a well established area in computer systems research. There are two issues related to energy management: energy consumption and peak power consumption. In battery-powered embedded devices, the energy consumption directly determines the lifetime of the system, and low-power components are needed in order to maintain the peak power (and thus the size of the cooling system) within acceptable limits. In high-end servers and server farms, energy management is equally important for different reasons. First, the electricity cost is already a significant fraction of the operation cost. Second, increasing performance demands and tighter packing results in high power densities and peak power values that require expensive cooling technologies.

Power management (PM) techniques can be classified as local and global. *Local PM* techniques put unused or underutilized local resources to low-power states. Examples include halting an idle processor, putting the memory chips to self-refresh and power-down states [29], or stopping the spinning of idle disks [55]. Typically only the highest power state is available for normal execution, with the notable exception of CPU dynamic frequency and voltage scaling (DVS) [94] common in most of today's microprocessors. *Global PM* refers to system-wide approaches for multiprocessor systems, such as turning on and off servers in a server farm, as required by the load [26]. A PM mechanism (local or global) is said to be *QoS-aware*

if it reduces the power consumption while guaranteeing the QoS, such as average response times or percentage of deadlines met.

Unfortunately, power management does not come for free, and should be used wisely. The main issue is the overhead of changing device state, which ranges from microseconds (DVS) to dozens of seconds for complete system reboot. Note there is also an energy overhead for switching a device from one power state to another. To avoid unnecessary overheads (such as oscillating too frequently between two device states), load prediction algorithms are typically used to estimate resource usage when performing power management decisions.

2.4 DYNAMIC VOLTAGE SCALING

Dynamic voltage scaling (DVS) is a local PM technique that allows performance-setting algorithms to dynamically adjust the voltage and frequency (and thus the performance) of the processor. An increasing number of processors implement DVS, which can yield quadratic energy savings for systems in which the dominant power consumer is the processing unit [44, 94], at the expense of just linear performance loss. In the presence of real-time constraints, DVS performance-setting algorithms attempt to lower the operating frequency and voltage while still meeting request deadlines.

For frame-based systems, the effects of having an upper bound on the voltage change rate are examined in [44]. Non-preemptive power aware scheduling is investigated in [42]. Slowing down the CPU whenever there is a single task eligible for execution was explored in [83]. For aperiodic tasks, Yao et al. [94] provided a static off-line DVS scheduling algorithm and on-line extensions, assuming worst-case execution times. Heuristics for on-line scheduling of aperiodic tasks while not hurting the feasibility of periodic requests are proposed in [43].

Cyclic and *EDF* scheduling of periodic hard real-time tasks on systems with two (discrete) voltage levels have been investigated in [47]. The static solution for continuous voltages and the general periodic model where tasks have potentially different (convex) power characteristics is provided in [3]. An interesting fact is that the problem of minimizing the energy consumption for convex power functions and no reward management is equivalent to the problem of maximizing the rewards for concave reward functions and no energy manage-

ment [4].

While static solutions rely on pessimistic WCET, much research has been directed at dynamic slack-management techniques [59, 4, 82, 63]. Moreover, DVS techniques can use workload estimations to further enhance the energy savings. For example, the expected energy consumption can be improved by using the probability distribution function of task execution times. This results in a DVS schedule that gradually increases the speed of executing tasks [54, 36, 95].

Schemes that adjust the speed based on system utilization are proposed in [88]. Another example of the utilization policy is Transmeta's firmware implementation (LongRun) [21]. CPU utilization is frequently monitored, resulting in performance speed-up/slow-down by one performance level. A software utilization-based DVS algorithm for personal computers running Linux [32] was shown to achieve 11%-35% more performance reduction over LongRun by using improved prediction and distinguishing between background and interactive jobs. Synthetic utilization bounds [1] have been proposed as a mechanism to provide delay guarantees in servers. For web servers, a DVS scheme based on such utilization bounds was described in [79]. Another utilization-based scheme for web servers was proposed in [10], which increases or decreases the CPU speed according to utilization thresholds.

In practice, reducing the processor performance may negatively affect the energy consumption in other system components, an aspect which is sometimes ignored in DVS research. For example, reducing the processor performance by half may require the memory subsystem to be used twice as long, increasing the overall energy consumption of the system. The interplay of DVS with memory power management for a particular system was studied in [29], with the conclusion that a unified power management is better than having separate DVS and memory power management policies. Acknowledging the importance of system power consumption (as opposed to considering just the CPU dynamic power in voltage scaling decisions), various system power models were recently proposed in the literature [96, 41, 26].

Moreover, although in theory the voltage can be scaled linearly with the frequency, many DVS processors do not have such a linear relationship, which results in some cases in inefficient operating frequencies. A frequency is inefficient if there exists a higher frequency that results in lower energy consumption. Any DVS algorithm should avoid using these

frequencies. An interesting fact is that the inefficient frequencies are not an artifact of only the processor characteristics, but also of the power consumption and management of other system components. Identifying the inefficient operating frequencies was studied in [58].

2.5 POWER MANAGEMENT IN SERVER FARMS

With energy consumption emerging as a key aspect of cluster computing, much recent research has focused on PM in server farms. There are two issues related to the power management of such systems, peak power control (to alleviate cooling problems) and energy management (to reduce electricity costs).

Lots of recent research has targeted the cooling problem. Trigger mechanisms for dynamic thermal management of high-performance microprocessors are investigated in [11]. For server farms, software-based temperature emulation and management techniques are proposed in [39]. A combined software/hardware technique is proposed for peak power control in high-performance servers, targeting the reduction of the cooling system size [30].

Addressing the energy problem, a first characterization of workloads and energy consumption in real-world web servers was made in [10]. DVS was proposed as the main technique to reduce energy consumption in such systems. Utilization-based DVS and request batching techniques were further evaluated in [79, 27].

The problem of *cluster reconfiguration* (i.e., turning on and off cluster machines) for homogeneous clusters was first addressed in [64]. An offline algorithm determines the number of servers needed for a given load. Cluster reconfiguration is then performed online using thresholds to prevent too frequent reconfigurations, even though there is no explicit QoS consideration. The authors have extended their work to heterogeneous clusters in [40], adding models for throughput and power consumption estimation. Reconfiguration decisions are made online based on the precomputed information and the predicted load. The authors also proposed to add request types to improve load estimation in [41].

Note that due to the convex nature of the power consumption function, load balancing across the processors minimizes the energy consumption in a homogeneous system. The assignment of tasks to processors is studied in [6]. One of the first attempts to combine

cluster-wide and local PM techniques for homogeneous clusters [26] proposed five policies combining DVS and cluster reconfiguration. Another work proposed to use the cluster load (instead of the average CPU frequency) as the criteria for turning on/off machines [93]. All approaches rely on power models and offline precomputed tables that determine the transition points when servers are turned on/off.

2.6 WORKLOAD ESTIMATION

Workload monitoring and prediction plays an important role in this work. Since the power consumption of systems increases more than linearly with the performance offered, workload monitoring and prediction allows systems to reduce the energy consumption by adjusting their performance to the minimum that can satisfy the predicted demand.

Traditional hard real-time application models (such as periodic or frame-based) assume that the arrival time, deadline and worst-case execution time (WCET) of real-time jobs are known a-priori. Such systems have to guarantee that real-time jobs are allotted enough resources to meet their deadlines even in worst-case scenarios. In practice however, real-time applications exhibit a large variation in actual execution times and WCET is too pessimistic. Without an explicit prediction of future workloads, the slack (system idleness) created by jobs that did not require their worst-case can be used to reduce the energy consumption of the system, while still providing real-time guarantees for subsequent jobs.

If the hard real-time constraint is relaxed, prediction of expected execution times can further reduce the energy consumption [48]. More advanced power management techniques use workload predictions to provide resources for the average case and gradually increase the performance to ensure that hard real-time constraints are met [54, 36, 95]. The prediction is typically based on the cumulative probability distribution function of task execution times. Request type information obtained from the request headers (such as static or dynamic pages for web servers) can further improve the prediction quality [41].

In most real-life situations, there is no prior knowledge of task arrival times and resource requirements. The workload is rather unpredictable in a variety of systems, from simple cell phones and PDA devices to more complex personal computers, servers and systems-on-a-

chip [49, 80]. For such systems (typically soft real-time), estimating arrival rates and request processing requirements are two important issues. Chandra et al. [16] applied a first-order autoregressive process to predict arrival rates and a histogram method to estimate the service demand.

Generally there are two ways of estimating the distribution from a sample: parametric and nonparametric [25]. Nonparametric methods are more suitable for unpredictable workloads than parametric methods. Govil et al. [34] did a comparative study of several predictive algorithms for dynamic speed-setting of low-power CPUs and concluded that simple algorithms based on rational smoothing rather than "smart" predicting may be most effective.

3.0 SYSTEM MODELS AND PROBLEM DESCRIPTION

This chapter introduces the models and notation used throughout the dissertation. Sections 3.1 through 3.3 introduce the three dimensions considered in this dissertation: deadlines, rewards and energy. Section 3.4 identifies the problems to be solved and gives a general research overview. Section 3.5 discusses the evaluation methodology.

3.1 REAL-TIME APPLICATION MODELS

3.1.1 Frame-based and Periodic Task Sets

For hard real-time systems we focus on frame-based and periodic task models. The task set is denoted by $\mathbf{T}=\{T_1, T_2, \ldots, T_N\}$, where N is the number of tasks. In the frame-based model all tasks share the same arrival time (the start of a frame) and deadline D, relative to the start of the frame. The deadline typically corresponds to the start time of the next frame. This model corresponds to real-time systems that operate in a cyclic basis, with a set of applications that must execute in a frame, whose execution is to be repeated. Examples of such applications include multimedia and real-time communication. The execution time of a task T_i depends on the number of cycles C_i and the task frequency s_i, and is given by $t_i = \frac{C_i}{s_i}$. The task set is schedulable if $\sum_{i=1}^{N} t_i \leq D$.

The periodic task model is a generalization of the frame-based model in which each task has its individual deadline d_i. The utilization of a task T_i is defined as $U_i = \frac{t_i}{d_i}$. The system utilization is defined as $U = \sum_{i=1}^{N} U_i$. For many scheduling algorithms, U serves as a schedulability bound. For example, under *EDF* scheduling [51] the task set is schedulable if and only if $U \leq 1$, while under *RMS* scheduling [51] a sufficient condition for schedulability

is $U \leq N(2^{1/N} - 1)$.

3.1.2 Aperiodic Request Servers

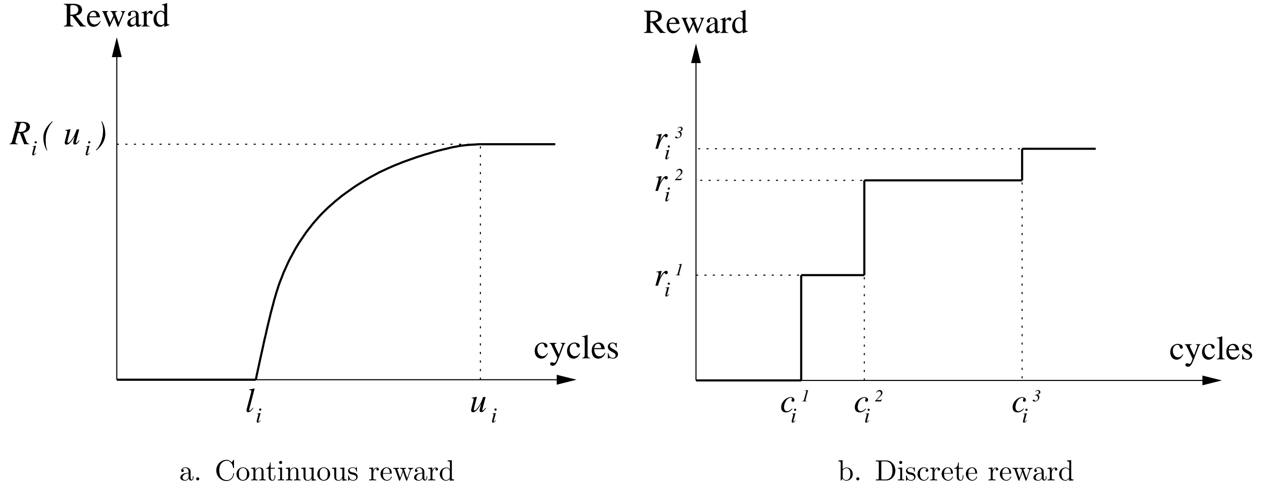
An aperiodic real-time model corresponds to irregular task arrival times. Upon arrival, each task is expected to complete by its deadline d_i. In addition, task execution times are highly variable. In practice, task inter-arrival times and execution times can be described by a distribution function or histogram. Due to unpredictability, the hard real-time constraints are relaxed and occasional deadline misses are tolerated. Scheduling can be either real-time (such as *EDF*) or can combine real-time constraints (for example, utilization-based DVS) with legacy schedulers (e.g., the Linux scheduler). The system reward is typically described in terms of QoS requirements, such as maintaining deadline miss ratios below certain thresholds.

3.2 REWARD MODELS

For hard real-time adaptive applications, the continuous and discrete reward models are presented next. The continuous reward model closely follows the IC (Imprecise Computations) model. For the discrete case, step reward functions are proposed, with no reward for partial execution. For soft real-time servers executing non-adaptive requests, the reward model is relaxed into a QoS requirement.

3.2.1 Continuous Reward Functions

Each task T_i is composed of a mandatory part M_i and an optional part O_i, with computational requirements expressed in terms of number of CPU cycles. The execution time depends on the number of cycles and the speed expressed in cycles/second (or the clock rate), $t_i = \frac{C_i}{s_i}$. The worst-case number of cycles needed for the mandatory part is denoted by l_i and the total worst-case number of cycles including the optional part is denoted by u_i. In other words, each task must receive a number of cycles between the lower bound l_i and the upper bound u_i.



a. Continuous reward   b. Discrete reward

Figure 3.1: Typical reward functions

The optional part of a task can execute only after its corresponding mandatory part completes. There is a reward function associated with each task; the reward function increases with the number of cycles that were allocated to the optional part. This reward is zero when the service time of the optional part is zero and increases with the amount of service time. There is no *extra* reward if the optional part receives more cycles than required (i.e., $u_i - l_i$). We consider the case of positive non-decreasing concave reward functions (the 2nd order derivatives of the reward functions R_i are negative). Note that for convex or partially convex reward functions, even assuming an infinite energy budget, determining the optimal solution that maximizes the total reward is NP-hard [5]. Figure 3.1a shows a typical reward function, which is described by:

$$R_i(C_i) = \begin{cases} 0 & \text{if } 0 \leq C_i \leq l_i \\ f_i(C_i) & \text{if } l_i \leq C_i \leq u_i \\ f_i(u_i) & \text{if } C_i \geq u_i \end{cases}$$

3.2.2 Discrete Rewards and Task Versions

For adaptive application domains that do not reward partial execution we propose a discrete model in which each task has several completion points, each with different rewards, time and energy requirements. This corresponds to step reward functions, and also applies to

applications that have several versions with different accuracy levels. For example, one version may require less cycles and therefore use less energy, at the expense of producing less accurate/complete/valuable results.

Real-life examples of task versions include satellite-based signal processing [84] (with different algorithms providing different levels of accuracy) and Automated Target Recognition (ATR) [37] (where task values and running times are roughly proportional to the number of targets detected). Task versions also correspond to unmodified applications with different input arguments (such as encoding/decoding at different QoS levels [89, 90], low/high quality compression schemes, low/high resolution image processing), or even different invocation periods, as proposed in [14]. For the remainder of this dissertation, the term "versions" also refers to discrete completion points of the same application (i.e., the discrete equivalent of the IC model).

For simplicity of notation, we assume the same number of versions V for each task, although the reward maximization algorithms proposed in this dissertation can handle different number of versions. The version k of task i is denoted by T_i^k. The execution time and energy requirement of version k of task i running at speed level j are denoted by $t_{i,j}^k$ and $e_{i,j}^k$ respectively. Associated with version k of task i there is a version value or reward, r_i^k. Figure 3.1b shows a typical step reward function corresponding to task versions (c_i^k denotes the number of cycles required by version k). For non-adaptive tasks (that is, $k = 1$, corresponding to all-or-nothing reward) the superscript k is omitted in the notation. Note that the reward maximization problem was shown to be NP-Complete for step reward functions [53].

3.2.3 Quality of Service for Aperiodic Requests

Many real-life application domains (especially aperiodic request servers, such as web servers) are non-adaptive, meaning that there is no reward function associated with requests. In some cases the utility of a request may depend on its response time (corresponding to time utility functions [50]), but typically the system reward/utility is defined in terms of simple QoS requirements, such as maintaining average response times and/or the percentage of deadline misses below given thresholds.

When system utility is defined as QoS, the problem is not to maximize the utility but rather to maintain a required QoS with minimal system costs (such as power consumption). For such application domains we focus mostly on soft real-time systems, where each request is viewed as an aperiodic task and with a soft deadline. In particular, power management opportunities that ensure the QoS are investigated, where QoS is defined as a percentage of requests that should meet their deadlines (without dropping any other requests). These opportunities are studied for both isolated systems (single server) and server clusters. These systems are typically underutilized (nowadays server farms are designed for peak load, with an average utilization of 10% to 20% [26]), which creates the opportunity for dynamic and QoS-aware system reconfiguration.

3.3 POWER MODELS

The power consumption for CMOS based processors is dominated by dynamic power dissipation P_d, which is given by [13, 17]:

$$P_d = C_{ef} \cdot V_{dd}^2 \cdot f \tag{3.1}$$

where C_{ef} is the effective switch capacitance, V_{dd} is the supply voltage and f is the processor clock frequency. There is an almost linear relationship between the frequency f and the supply voltage V_{dd}, given by:

$$f = k \cdot \frac{(V_{dd} - V_t)^2}{V_{dd}} \tag{3.2}$$

For the continuous model in Section 3.3.1, the power function is derived from Equation 3.1 and is described as a function of the processor speed. The continuous power function currently only models the processor power consumption, although a constant power for other system components can be easily included. The discrete case in Section 3.3.2 corresponds to discrete processor frequencies/voltages. The discrete energy consumption values represent the total energy consumed by a task at a given frequency (note that energy is defined as the integral of power over time) and may refer either to the processor (as described by Equation 3.1) or the entire system. Finally, a more detailed system power model with the corresponding notation is introduced in Section 3.3.3.

3.3.1 Continuous Speeds/Voltages

The speed of task T_i (denoted by s_i) is physically constrained to be within certain lower and upper bounds S_{min} and S_{max}. The continuous model assumes that s_i can take any value in the given range (that is, voltages and frequencies can be changed continuously and $S_{min} \leq s_i \leq S_{max}$). Typically, the speed is normalized (that is, $S_{max} = 1$).

One assumption is that each task T_i runs at single speed s_i and for each task T_i there is a corresponding power function P_i, which depends on its speed and on the switching activity of the task. Inspired by Equation 3.1, we consider the realistic case of positive non-decreasing convex power functions (the 2nd order derivatives of the power functions P_i are positive). Thus, it is more energy efficient to run on reduced speeds, at the expense of linear performance loss. In this dissertation, the term *speed change* refers to changing both the frequency and the supply voltage. For this model, we assume that the worst-case time and energy overheads of speed transitions was already accounted for (for example, at most two speed transitions per task are necessary with *EDF*, at task arrival and task completion).

The energy consumed by task T_i running for time t_i at speed s_i is $E_i = t_i \cdot P_i(s_i)$. It is assumed that there is a limited energy budget in the system, denoted by E:

$$\sum_{i=1}^{N} t_i \cdot P_i(s_i) \leq E \tag{3.3}$$

3.3.2 Discrete Speed/Voltage Levels

In the discrete model, the variable voltage processor has only a discrete number, M, of available frequencies (clock rates or CPU speeds), $\{f_1, f_2, \ldots, f_M\}$. Each task can run at any of the available speeds and we say that a task runs at speed level k if the speed of the task is set to f_k.

For hard real-time models we assume that the task worst-case execution time and energy consumption values are known for all task versions and all speed levels (that is, all $t_{i,j}^k$ and $e_{i,j}^k$ are known). As in the continuous model, the total energy consumed by the system cannot exceed the energy budget E.

3.3.3 System Power Model

In the continuous energy model, the power function considers only the processor. With small modifications, a fixed cost for other system components may be easily incorporated. In the discrete case, the energy values, $e_{i,j}^k$, can refer to either the processor or the entire system energy consumption.

However, for certain systems, the above continuous and discrete models may be impractical or incomplete. The power functions $P_i(s_i)$ (for the continuous case) or the energy values $e_{i,j}^k$ (for the discrete case) are typically not available for soft real-time systems with unpredictable workloads. In addition, the models are incomplete for systems where the idle system power plays an important role (both continuous and discrete models derived from Equation 3.1 assume that no power is consumed when the system is idle). Furthermore, for multiprocessor systems or server farms, unused systems may be put to even lower power, or completely shut down. Thus, a complete model has to include the power consumption when the system is idle or even off ("off" power is not necessarily zero if hardware is needed to wake up the system on certain events, such as Wake-on-LAN packets).

The following notation is introduced: the power of a system actively processing requests at frequency f_i is denoted by P_i, the power of an idle system is denoted by P_{idle}, and the power consumption when the system is off is denoted by P_{off}. Obviously, P_i depends on the operating frequency. P_{idle} may or may not depend on the frequency, depending of the type of idle power management. We also use the notation P_{load} to denote the average power consumption of a system as a function of its load. P_{load} depends not only on the load, but also on the scheduling policy, various system parameters (such as bus speeds, disk throughput and memory size), and the power management schemes in place (including but not limited to DVS). Thus P_{load} is the most general case of system power function, and considers all system parameters. P_{load} is often hard to derive theoretically and is determined through system power measurements on representative traces. The determination of P_{load} for real-life servers will be discussed in detail in Chapter 4.

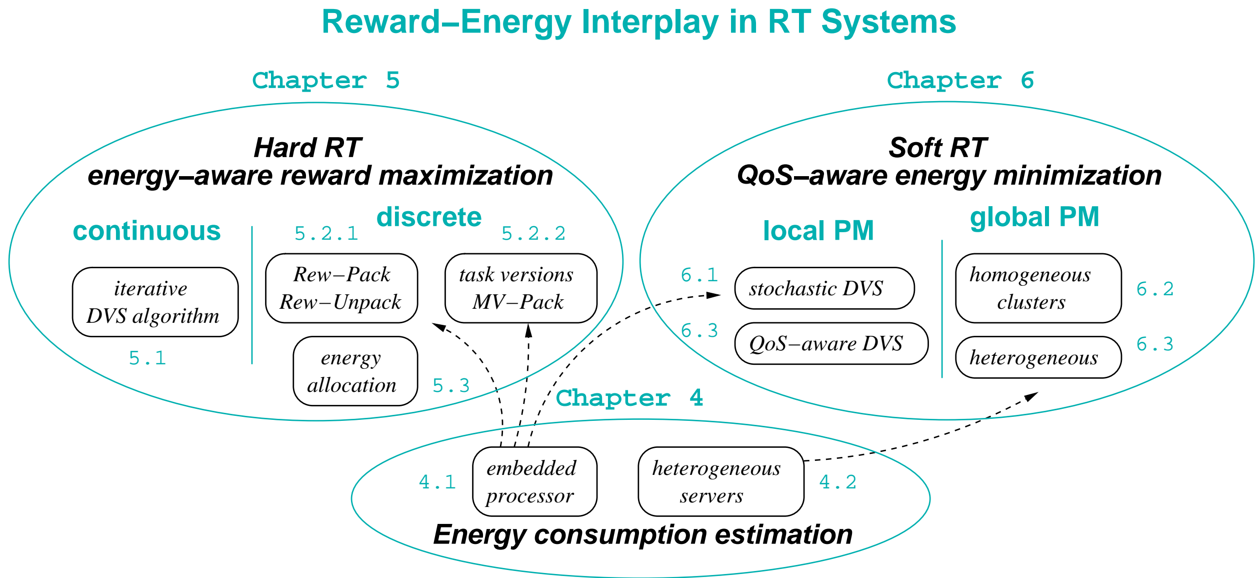


Figure 3.2: Research overview

In this dissertation, the interplay of reward and energy is studied in the context of real-time systems. While previous work focused on only two of the constraints at the same time (either reward maximization or energy minimization given real-time constraints), we focus on reward maximization techniques for systems with fixed energy budgets and energy minimization techniques for systems with fixed QoS/reward requirements respectively. The reward/energy interplay is studied for a variety of systems (ranging from embedded to server farms) and real-time application models (both hard and soft real-time). A general overview of our research is given in Figure 3.2.

For hard real-time systems (Chapter 5) we focus on frame-based and periodic applications. Continuous reward and power models are first assumed in Section 5.1. Extending a reward-maximization algorithm [5] to also incorporate a fixed energy budget, the problem is to maximize the system reward while guaranteeing real-time and energy constraints. An iterative DVS algorithm is proposed and shown to closely approximate the optimal.

The realistic case of discrete reward and power models is studied in Section 5.2, for for both frame-based and periodic application models. Two algorithms are proposed for re-

ward maximization of non-adaptive (i.e., all-or-nothing) applications, *REW-Pack* and *REW-Unpack*, and shown to have worst-case running times in the microsecond range, with results within 3% of the optimal (on average) in our experimental setup. An extension of the reward model is then proposed, in the form of task versions, corresponding to the general case of step reward functions. A third algorithm, *MV-Pack*, is devised as an extension of *REW-Pack* for multiple versions, with similar running times and results.

The above reward maximization algorithms are short-term, as they assume a fixed energy budget over a relatively short fixed period of time. Long-term reward maximization issues are further addressed in Section 5.3. The allocation of energy budgets in the long run is first discussed in Section 5.3.1 for the simple case of battery-powered embedded systems. Extending to systems that rely on rechargeable energy, static and dynamic policies for allocation of energy budgets are investigated in Section 5.3.2. Three dynamic policies are shown to outperform a static conservative scheme, by using various energy reclamation and prediction mechanisms to further improve the long-term system reward.

For soft real-time systems running non-adaptive applications, the reward model is relaxed into a QoS constraint (Chapter 6), and the optimization objective is the energy consumption. We first investigate local power management techniques (DVS policies) for unpredictable workloads (Section 6.1). A stochastic soft-real time DVS policy that uses the cumulative distribution function of task execution times is shown to largely outperform (in terms of total energy consumption) utilization-based and prediction-based DVS policies. Similarly, a local QoS-aware DVS scheme is proposed for power management in web servers (Section 6.3).

For multi-processor soft real-time systems, global QoS-aware power management policies are first investigated for homogeneous real-time clusters (Section 6.2). The policies rely on offline precomputed tables that determine the number of servers needed as a function of system load. The work is then extended to the general case of heterogeneous clusters in Section 6.3. An integrated local (DVS) and global (on/off) scheme relying on measurement-based power modeling is shown to achieve significant energy savings while maintaining a desired QoS.

We note that while application deadlines and reward functions are known a-priori, the third dimension studied in this dissertation requires estimation of the energy consumption

of tasks. Chapter 4 discusses power/energy estimation techniques for both embedded and high-performance systems. A fine-grained event-based power model for PowerPC embedded processors is derived in Section 4.1. For high-performance servers, a coarse-grained measurement-based approach is used to identify the system power from load observations (Section 4.2).

3.5 EVALUATION METHODOLOGY

The solutions proposed in Chapter 5 for hard-real time application models are evaluated against existing schemes mainly through simulations. Task sets were artificially generated, with realistic execution times and deadlines and a broad range of task rewards/values. The continuous power model uses a variety of convex power functions derived from Equation 3.1, as well as more complex and realistic functions [48] (Section 5.1). For the discrete case, the proposed schemes (Section 5.2) were studied on a variety of power models, including Intel XScale [57] and PowerPC405LP. The exact frequencies, voltages and power values used are indicated throughout the thesis. For PowerPC systems we use IBM's Mambo simulator [78] to provide cycle accurate timing and power consumption information. The derivation and validation of an event-based power model for PPC405GP and PPC405LP processors is described in Section 4.1.

For homogeneous soft real-time clusters, the proposed schemes in Section 6.2 are evaluated on real traces of signal processing applications provided by our research partners. The global and local power management schemes are implemented in a PPC750-based experimental platform. The complete trace characteristics, as well as the measured frequencies and system power consumption values (P_i, P_{idle}, P_{off}), are given in Sections 6.1 and 6.2. For the general case of heterogeneous soft real-time clusters (Section 6.3), QoS-aware power management techniques are implemented in an Apache [2] web server cluster. Trace characteristics and the measurement-based derivation of the general system power function P_{load} for heterogeneous servers is discussed in detail in Section 4.2.

The proposed algorithms are evaluated against existing algorithms, no-power-management schemes, and optimal solutions (where such a comparison was possible). In some cases, the

solutions are compared against an upper bound on the optimal solution. The metric used for comparison is the overall system reward for reward maximization schemes. For experiments with servers and unpredictable workloads (no task rewards) the metric is the energy consumption augmented with QoS considerations.

4.0 POWER CONSUMPTION ESTIMATION

In this work the interplay of reward and energy in the context of real-time systems is analyzed. Application deadlines and reward functions are known a-priori. The energy consumption, however, must be estimated either from power models (such as Equation 3.1), published data (e.g, XScale [57]) or from our own power measurements. In this chapter two approaches are presented that combine models with direct measurements, in order to better understand the power/energy consumption of the applications considered in this dissertation.

First, a fine-grained event-based power/energy estimation technique is proposed for PowerPC405 (PPC405GP and PPC405LP) embedded processors. The resulting power model was validated and incorporated into IBM's cycle-accurate Mambo simulator for PowerPC systems [9]. The Mambo simulation infrastructure was then used to determine power consumption values for PPC405LP embedded processors. The derived values are used for the evaluation of DVS algorithms on the PPC405LP platform in Section 6.1 (Table 6.3), as well as for the evaluation of reward maximization algorithms in [70].

Second, a coarse-grained measurement-based approach for the power modeling of high-performance servers is analyzed. In contrast with the PPC405 power estimation (which identifies certain energy events with fine granularity) this is mostly a measurement-oriented technique. The system power consumption (of high-end servers in our case) is determined through offline measurements of representative benchmarks for various system loads. In other words, we are determining through experimentation the general power function P_{load} introduced in Section 3.3.3. The derived power functions P_{load} for a variety of heterogeneous servers are then used in Section 6.3 as the basis for power-aware request distribution and global power management techniques for heterogeneous server clusters.

4.1 FINE-GRAINED CPU POWER MODEL

In this section we present an event-based approach for power modeling of PPC405 embedded processors. Events of interest for power are identified, corresponding to various microarchitectural events, such as cache misses or ALU operations. The energy costs are then obtained through power measurements of specially designed microbenchmarks (small assembly loops). The PPC405 architecture is first introduced, followed by a description of the power events used, and validation.

4.1.1 The PPC405 Architecture

The PPC405 core is a relatively simple architecture due to its in-order issue. The PPC405 core implements the PowerPC instruction set using a simple five-stage pipeline, and no floating point unit, as shown in Figure 4.1. Most instructions execute in one cycle, but some instructions (such as multiplication or division) require more cycles. Furthermore, some functional units are pipelined and load misses can overlap with independent instructions. The processor core frequency used in our experiments is 200MHz, at core voltage 2.5V. The processor local bus (PLB) is running at 66MHz, and the I/O subsystem uses 3.3V.

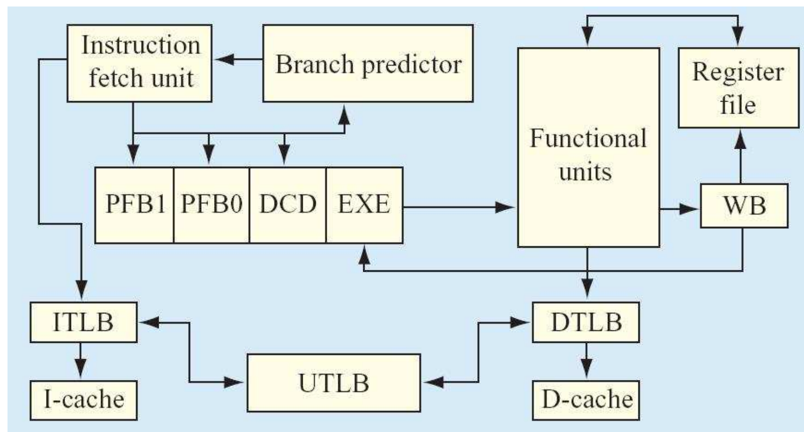


Figure 4.1: The PPC405GP pipeline

The PPC405GP processor (used in our evaluation) does not support voltage scaling. Another processor of the same family (the PPC405LP) uses the same core and also supports

voltage scaling. Thus, the same methodology (described in the next section) can be used to identify the power models for both 405 embedded processors.

4.1.2 Power Modeling Events

We use a relatively simple power model, in which the total energy consumed E_{total} is the sum of the static (or idle) energy consumption E_{idle} and the energies e_i consumed by various microarchitectural events. Thus, if n_i denotes the number of times event i occurs, the total energy is given by

$$E_{total} = E_{idle} + \sum_i n_i e_i \tag{4.1}$$

The events of interest for power consumption were identified through experimentation. An experimental 405GP board with separate power planes was used to measure the processor power consumption. To determine the event energy costs we developed over 300 microbenchmarks (small assembly loops) emphasizing particular events. An example of such microbenchmark is given below:

```
loop:
lwz r2, 0 (r3)
addi r2, r2, 1
lwz r2, 0 (r4)
addi r2, r2, 1
lwz r2, 0 (r5)
addi r2, r2, 1
bctr loop
```

The above microbenchmark is used for measuring the timing and energy consumption of a data cache miss. Data is used immediately after being loaded from memory, causing processor stalls if the data is not in the data cache. There are three loads from the addresses stored in registers *r3*, *r4* and *r5*. If the registers contain the same address there are no cache misses (except for the first compulsory miss). When the addresses are different but map to the same cache set, every load in the microbenchmark results in a miss for a two-way associative cache (as is the case with the PPC405 architecture).

After measuring the timing and energy consumption, the extra time/energy required can be attributed to the time penalty and the energy cost of a data cache miss (relative to a hit)

respectively. Let N denote the loop count, C_{hit} and P_{hit} denote the number of cycles and power when the loop has no misses, and C_{miss} and P_{miss} denote the corresponding values for the case of cache misses. Similarly, T_{hit} and T_{miss} denote the execution time (that is, cycles multiplied with cycle time, which is 5ns in our case since the CPU runs at 200MHz). Then, the data cache miss penalty (in cycles) is $(C_{miss} - C_{hit})/3N$ (note that $3N$ is the number of loads, i.e., cache misses), and the data cache miss extra energy (relative to a data cache hit), $E_{dcachemiss}$, can be determined from the equation

$$P_{miss}T_{miss} - P_{hit}T_{hit} = (T_{miss} - T_{hit})P_{idle} + 3NE_{dcachemiss} \tag{4.2}$$

where P_{idle} is the measured idle power.

Note that in this example, loads replace only clean lines, so there are no line writebacks. In addition, since loads are followed by uses, the pipeline stalls until the loads return, hence the term involving P_{idle}.

Similar microbenchmarks and equations were derived for the other power events. For example, to measure instruction energy costs, we based our measurements on a loop of NOP instructions. We then added the instruction of interest to the loop and measured the contribution of that instruction/functional unit to power consumption.

A special event called *average instruction switch* is introduced to reflect the energy consumed by the pipeline and control path of the processor as different instructions progress through the core. This event resulted from the observation that a large variation in power occurs when reordering instructions within a loop (without affecting functionality and running time). For example, a loop with six loads and six independent adds consumes 15% less power/energy if the loads are all executed consecutively followed by all of the adds compared with the same loop when alternating between loads and adds. Since modeling such switching is beyond the scope of this work, we chose instead to use an average value based on a limited number of experiments.

The complete lists of events is shown in Table 4.1 and includes the average energy cost of cache hits/misses, various instruction types, branch conditions, TLB hits/misses, etc.

Table 4.1: Power modeling events

Event	Description
CPU static energy	Base energy for every simulated cycle
Switching	Average energy due to switching in pipeline
NOP	NOP instruction (additional base energy in active state)
ALU	Logic, addition, subtraction, move, etc. instructions
Load/store	Load/store instructions
Divide/multiply	Divide/multiply instructions
PFB0 branch	Branch instruction placed in PFB0 buffer
DCD branch	Branch instruction placed in decode stage
Mispredicted branch	Branch misprediction (flushing pipeline)
ITLB miss, UTLB hit	ITLB miss satisfied by the UTLB
DTLB miss, UTLB hit	DTLB miss satisfied by the UTLB
TLB read	tlbrehi or tlbrelo instruction
TLB search	tlbsx instruction
TLB write	tlbwehi or tlbwelo instruction
TLB sync	tlbsync instruction
I-cache hit	I-cache hit to same cache line as before
I-cache hit other	I-cache hit to another cache line
I-cache miss	I-cache miss
D-cache hit	D-cache load/store hit to same cache line as before
D-cache hit other	D-cache load/store hit to another cache line
D-cache miss	D-cache miss
D-cache line flush	D-cache replacement causes a writeback

4.1.3 Power Modeling Validation

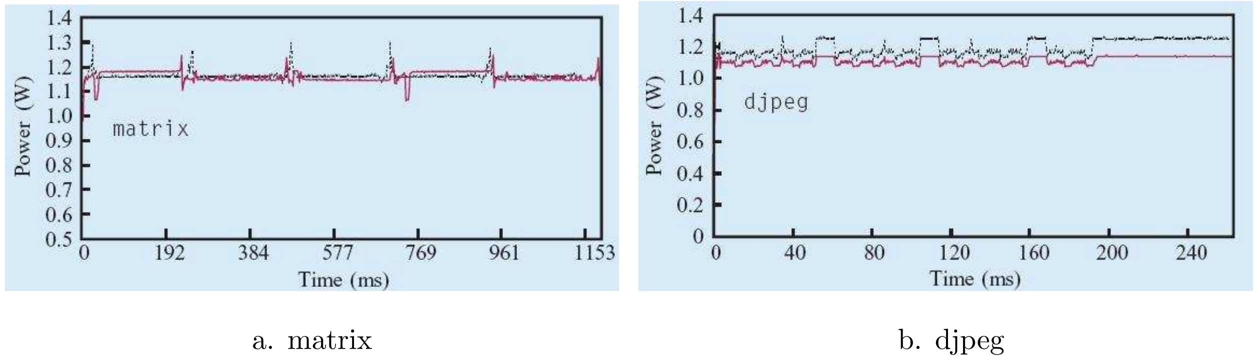
The power model was incorporated in IBM's cycle accurate Mambo simulator [9] for the PPC405 architecture. The additional simulator overhead of tracking energy events is negligible, since it required only the addition of an energy counter. This counter is incremented as appropriate during simulation runtime, with the energy costs corresponding to either instructions (an energy value was added to the information structure of each instruction) or microarchitectural events (such as cache misses).

As benchmarks we use 39 Embedded Microprocessor Benchmarking Consortium (EEMBC) benchmarks [20]. The results for all benchmarks are presented in Table 4.2. The third column shows the actual energy (in millijoules), as measured on our 405GP experimental platform. The maximum absolute error in energy consumed is 11.3%, while the average absolute error is just 5.9%. Note that for the same benchmarks the error in execution time was at most 7.1% and just 2% on average.

Table 4.2: Energy validation results

Benchmark	Simulated energy (mJ)	Actual energy (mJ)	Error (%)
a2time	2.50	2.40	4.44
aifftr	807.52	819.40	-1.45
aifirf	4.19	4.36	-3.99
aiifft	734.07	752.80	-2.49
autcor (pulse)	1.62	1.70	-4.67
autcor (sine)	241.05	258.40	-6.72
autcor (speech)	230.02	245.80	-6.42
basefp	19.44	18.49	5.12
bezier	522.69	502.30	4.06
bitmnp	84.33	81.30	3.73
cjpeg	342.15	354.50	-3.48
conven (k3)	82.28	87.12	-5.55
conven (k4)	104.95	111.60	-5.96
conven (k5)	120.67	127.80	-5.58
dither	1238.39	1291.00	-4.07
djpeg	296.86	311.90	-4.82
fbital (pent)	289.11	318.50	-9.23
fbital (step)	29.82	32.89	-9.32
fbital (typ)	633.24	677.90	-6.59
fft (sine)	68.57	70.36	-2.55
fft (spn)	68.42	71.25	-3.97
fft (tpulse)	68.40	70.77	-3.34
filters	1335.55	1389.00	-3.85
idctm	91.19	92.38	-1.28
iirflt	4.17	3.99	4.48
matrix	1378.78	1298.00	6.22
ospf	80.83	88.16	-8.31
pktflow	539.48	593.30	-9.07
pntrch	31.95	33.64	-5.03
rgbcmy	1257.79	1369.00	-8.12
rgbyiq	1332.43	1394.00	-4.42
rotate	662.80	701.80	-5.56
routelookup	252.91	283.50	-10.79
tblook	8.37	7.85	6.61
ttsprk	3.15	3.30	-4.66
Viterbi (gett)	254.85	285.30	-10.67
Viterbi (ines)	254.85	286.10	-10.92
Viterbi (toggle)	254.73	287.20	-11.30
Viterbi (zeros)	254.70	284.10	-10.35

Finally, in Figure 4.2 we show the simulated versus measured power consumption for two EEMBC benchmarks. The sampling period for both the simulator and the hardware was set to 20K cycles. Clearly, the model closely follows the measured power corresponding to various application phases.



a. matrix

b. djpeg

Figure 4.2: Hardware versus simulated power graphs

4.2 COARSE-GRAINED SYSTEM POWER MODEL

The technique presented in the previous section is limited to the power/energy estimation of a simple in-order embedded processor, ignoring other system components, such as memory or network. This section evaluates a measurement-based approach to the power modeling of entire systems, including hardware as well as software components. While the event-based approach is fine-grained (for example, the power consumed by caches or various functional units can be estimated), the approach evaluated in this section is coarse-grained, providing only a system-wide power/energy estimation (that is, the power breakdown of various system components is unknown).

The main idea behind the measurement-based technique is to estimate the system power through offline measurements for various system loads. Measurement-based approaches have several advantages. First, they are simple, as there is no need for models (except for load estimation). Second, they are accurate, provided that load is defined correctly and representative benchmarks are used for measurements. Third, the entire system power is accounted for, including the interaction of various hardware and software components. These advantages come at the expense of offline overheads: representative benchmarks have to be identified for power measurements, and actual measurements are needed in order to determine the system power consumption for various values of the load. Moreover, changing any system parameter (such as DVS scheme or adding more memory) requires redoing the measurements. Note, however, that the measurement process can be automated, effectively reducing the overheads.

Power Measurements of Heterogeneous Servers The measurement-based approach is evaluated for heterogeneous servers, with a specific case study of Apache web servers. This approach was motivated by our work in local and global power management of heterogeneous server clusters (Section 6.3). Our experimental cluster consists of five heterogeneous servers, as shown in Table 4.3. The server parameters include memory sizes, time to boot and shutdown, as well as the power when the server is off (note that this power is not zero, as the Wake-On-LAN interface needed for server reboot is still consuming power).

Table 4.3: Parameters of the machines of the Apache cluster

Machine name	Processor	RAM memory size	Cache size	Wake-On-LAN support	Boot time (sec)	Shutdown time (sec)	Off power (W)	Max load
Transmeta	Transmeta Crusoe TM5800	256 MB	512 KB		100	60	1	0.10
Blue	AMD Athlon 64 Mobile 3400+	1GB	1 MB	$\surd$	33	11	8	0.95
Silver	AMD Athlon 64 3400+	1GB	512 KB	$\surd$	33	12	8	1.00
Green	AMD Athlon 64 3000+	1GB	512 KB	$\surd$	33	11	8	0.90
Front-end	AMD Athlon 64 Mobile 3400+	1GB	1 MB	Not applicable				

Table 4.4: Idle/busy power consumption (in Watts) for each Apache server

Transmeta

Frequency (MHz)	333	400	533	667	733
Idle (W)	8	8.5	8.5	9	9
Busy (W)	9	9.5	10.5	12	12.5

Blue

Frequency (MHz)	800	1800	2000	2200
Idle (W)	68	73	76	80.5
Busy (W)	74.5	93.5	105.5	120.5

Silver

Frequency (MHz)	1000	1800	2000	2200	2400
Idle (W)	70	74.5	78.5	83.5	89.5
Busy (W)	80.5	92.5	103.5	119.5	140.5

Green

Frequency (MHz)	1000	1800	2000
Idle (W)	68	79	87
Busy (W)	77	108	131

The specific frequencies of each server and their corresponding power consumption are shown in Table 4.4. These values could be used as a power model for each server, provided that the percentage spent at each frequency is known. We choose, however, to go one step

further and determine the power of each server based on one single parameter: the server load. Thus, we are determining the system power function P_{load} introduced in Section 3.3.3. The last column in Table 4.3 shows the maximum load each server can handle. The determination of this value, as well as the P_{load} function is described next.

As observed in previous work [73, 93, 41], load estimation can be greatly improved by considering request types. The type of a request may be conveniently determined only by the header (e.g., the name of the requested file). In the case of a web server there are two main types of requests, with different computational characteristics: static and dynamic pages. Static pages reside in server's memory and do not require much computation. Dynamic pages, instead, are created on-demand through the use of some external language (e.g., Perl or PHP). For this reason, dynamic pages typically require more computation than static ones.

Consider a generic server, and let A_{static} and $A_{dynamic}$ be the average execution times to serve a static and a dynamic page, respectively, at the maximum CPU speed. For example, for one server in our cluster we measured an average execution time $A_{static} = 438\mu s$ for static pages and $A_{dynamic} = 24.5ms$ for dynamic pages. On average, the time needed by the CPU to serve N_{static} static requests and $N_{dynamic}$ dynamic requests is thus $N_{static}A_{static} + N_{dynamic}A_{dynamic}$ seconds. If the number of requests is observed over a period of $monitor_period$ seconds, then the load of the machine serving the requests is

$$Load = \frac{N_{static}A_{static} + N_{dynamic}A_{dynamic}}{monitor_period} \tag{4.3}$$

Notice that this definition of load assumes a CPU-bound server. This is normal for most web servers because much of the data are already in memory [10, 91]. In fact, on all our machines we have noticed that the bottleneck of the system was the CPU. However, for systems with different bottlenecks (e.g., disk I/O or network bandwidth) another definition of load may be more appropriate. In fact, the definition of load should account for the bottleneck resource. Even though web requests may exhibit a large variation in execution times, using the average values (A_{static} and $A_{dynamic}$) in Equation 4.3 results in a very accurate load estimation, because web requests are relatively small and numerous. We define the

maximum load of a server as the maximum number of requests that it can handle meeting the 95% of deadlines, which is our defined QoS requirement.

Once load is defined, representative benchmarks (web traces in our case) are collected for power measurements. The statistics of the traces are shown in Table 4.5, indicating the percentage of requests for various static page sizes and dynamic page running times, as observed in our webserver.

Table 4.5: Web server statistics: static page sizes and dynamic page running times

Request type	%	Request type	%
4 ms (CGI)	0.10	6-7 KB (html)	2.84
7 ms (CGI)	0.71	7-8 KB (html)	1.58
23 ms (CGI)	0.98	8-9 KB (html)	1.80
40 ms (CGI)	0.23	9-10 KB (html)	1.87
200 ms (CGI)	0.06	**10-20 KB (html)**	**10.74**
0-1 KB (html)	**37.78**	20-30 KB (html)	3.62
1-2 KB (html)	8.86	30-40 KB (html)	1.17
2-3 KB (html)	6.56	40-50 KB (html)	0.67
3-4 KB (html)	4.58	50-60 KB (html)	0.80
4-5 KB (html)	4.94	60-70 KB (html)	1.46
5-6 KB (html)	3.38	above 70 KB (html)	5.27

After the OS, the scheduling policy, and the local PM scheme (see Section 6.3.3.1) have been decided for a server, the power consumption as function of the load and the maximum load can be determined through simple measurements. The power consumption of our experimental servers is shown in Figure 4.3.

We measured AC power directly, with a simple power meter with 2% accuracy [76]. In our case, recording the average power consumption for a given load over a period of 10 minutes was sufficient to obtain a statistically significant average. The last point on each curve represents the maximum load that meets our QoS specification (i.e., 95% of deadlines met), normalized to the fastest machine in the cluster, namely *Silver*. Note that the power and performance (maximum load) values of *Transmeta* are much less than those of the Athlon servers.

The power functions P_{load} in Figure 4.3 represent our measurement-based model. Load is estimated dynamically based on Equation 4.3 and the power functions are used at runtime for power-aware load distribution. As a validation of our measurement-based approach, we

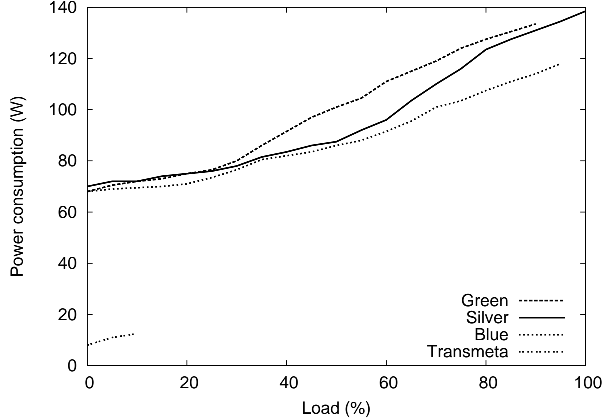


Figure 4.3: Power consumption versus load for Apache servers

note that the total energy of the cluster for real web traces (estimated using the measured power functions) was within 1% of the actual (measured) total cluster energy, as will be shown in the evaluation of cluster-wide power management schemes (Section 6.3).

4.3 CHAPTER SUMMARY

The focus in this chapter is on energy and power estimation techniques, corresponding to the bottom part of the research overview in Figure 3.2. Two approaches that combine models with direct measurements are evaluated.

A fine-grained event-based power model for embedded PowerPC processors is first discussed, relying on certain energy events with measured energy consumption. The model was incorporated into the Mambo [9] simulation infrastructure and validated against real hardware for the 405GP processor, shown to be within 6% of actual values on average, with a maximum error of 11.3%. The derived model is used for evaluation of DVS algorithms in Section 6.1, as well as reward maximization algorithms in [70].

The event-based model only includes processor power, ignoring the interactions of various hardware and software components. To study the effect of power management algorithms on the overall system power, we also evaluate a measurement-based coarse-grained (system-wide) power/energy estimation technique for servers. The technique determines the general

system power function P_{load} (introduced in Section 3.3.3) through direct power measurements. The definition of load, traces used, as well as the derivation of the power functions for a particular heterogeneous Apache web cluster are presented. The measurement-based approach is convenient, system-wide and accurate, at the expense of offline measurement overheads. Using the measured functions P_{load} as the basis for power management in our experimental Apache cluster (Section 6.3), the overall energy consumption is estimated within 1% of actual measurements for the web traces used in our experiments.

5.0 ENERGY-AWARE REWARD MAXIMIZATION FOR HARD REAL-TIME SYSTEMS

In this chapter we study reward-based scheduling algorithms for continuous reward and power functions, for hard real-time systems with a limited energy budget. While previous work targeted either reward maximization or energy minimization given task deadlines, this is the first work that combines all three constraints, namely energy, deadlines and task rewards.

Theoretical results and an iterative scheduling algorithm for continuous reward and power models are first described in Section 5.1. The study continues with more realistic discrete models for both power and reward functions. Task versions are proposed as an extension of the discrete reward model corresponding to step reward functions. Algorithms for the discrete reward maximization problem are presented in Section 5.2, for both single-version and multiple-version reward models. Finally, as the proposed algorithms maximize rewards given a fixed energy budget, we propose how to determine such energy budgets in the long run. In particular, we investigate long-term energy budget allocation policies for battery-powered embedded systems, as well as systems that use rechargeable energy sources in Section 5.3.

5.1 CONTINUOUS REWARDS AND POWER FUNCTIONS

This section presents our reward maximization algorithms for continuous power and reward models. The problem definition is first given, for both frame-based and periodic tasks. We then derive properties of the optimal solutions to the reward maximization problem. Based on these properties we identify optimal solutions for specific power functions, and derive an

iterative algorithm for general power functions.

5.1.1 Problem Definition

Using the notation introduced in Chapter 3 for the continuous case, the problem definition for frame-based tasks is presented next. Formally, for unknowns t_i and s_i (the execution time and the running speed for each task), the problem is the following:

$$\text{maximize} \quad \sum_{i=1}^{N} R_i(s_i \cdot t_i) \tag{5.1}$$

$$\text{subject to} \quad l_i \leq s_i \cdot t_i \leq u_i \tag{5.2}$$

$$S_{min} \leq s_i \leq S_{max} \tag{5.3}$$

$$\sum_{i=1}^{N} t_i \leq D \tag{5.4}$$

$$\sum_{i=1}^{N} t_i \cdot P_i(s_i) \leq E \tag{5.5}$$

Equation 5.1 is our maximization objective (i.e., the system value/reward). The system reward is thus defined as the sum of rewards R_i for all applications. Under the continuous reward model (Figure 3.1a), the reward of a task depends on the number of cycles C_i allotted to the task, $C_i = s_i \cdot t_i$. The restrictions of the IC model (i.e., the number of cycles allotted to the task) are present in Equation 5.2. The physical restrictions of the processor voltage scaling is given in Equation 5.3. The real-time schedulability condition for frame-based tasks appears in Equation 5.4. Finally, the energy constraint is found in Equation 5.5. The total energy consumed by all tasks cannot exceed an energy budget E.

For the more general case of periodic tasks, each task T_i has a different deadline d_i, and LCM denotes the least common multiple of all task periods. We introduce the notion of *supertask* T_i' to represent all instances of the same task T_i occurring during LCM time units. Note that the reward of periodic systems with limited energy budgets is maximized when all instances of a task run at the same speed and for the same amount of time, as shown in [69]. Based on this result, and noting that $\frac{LCM}{d_i}$ instances of task T_i execute in the LCM, the reward of a supertask T_i' is given by $R_i' = R_i \frac{LCM}{d_i}$ and its execution time is $t_i' = t_i \frac{LCM}{d_i}$.

The restrictions on the number of cycles of T_i' become $l_i' = l_i \frac{LCM}{d_i}$ and $u_i' = u_i \frac{LCM}{d_i}$. The problem then becomes to identify the parameters (speed s_i' and time t_i') of each supertask. Equations 5.1-5.5 for the periodic case become:

$$\text{maximize} \quad \sum_{i=1}^{N} R_i'(s_i' \cdot t_i') \tag{5.6}$$

$$\text{subject to} \quad l_i' \leq s_i' \cdot t_i' \leq u_i' \tag{5.7}$$

$$S_{min} \leq s_i' \leq S_{max} \tag{5.8}$$

$$\sum_{i=1}^{N} t_i' \leq LCM \tag{5.9}$$

$$\sum_{i=1}^{N} t_i' \cdot P_i(s_i') \leq E \tag{5.10}$$

The real-time schedulability condition 5.9 assumes *EDF* scheduling. Note that substituting the parameters of task T_i (R_i, s_i, t_i, l_i, u_i, D) of Equations 5.1-5.5 by the supertask T_i' parameters (R_i', s_i', t_i', l_i', u_i' and respectively LCM) we obtain Equations 5.6-5.10. Thus, the problem formulations for frame-based and periodic tasks are equivalent. For simplicity of notation, throughout the remaining of this section we use the notation for frame-based task sets. Clearly, all results (including complexity) apply to periodic tasks as well.

5.1.2 Properties of the Optimal Solution

A *solution* of 5.1-5.5 is defined as a set of values $\mathbf{S}$=$\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$ that satisfy the constraints 5.2-5.5 and also maximize the reward as described by 5.1. The value of this maximum reward is referred to as *optimal reward*. Because there may be more than one solution that maximizes the reward, a *minimum-energy solution* is defined as the solution consuming the least amount of energy. Some properties of the optimal solutions are presented next. For the results below, recall that u_i is the maximum number of cycles to be allotted to task T_i and let R_{ub} denote the maximum achievable reward (i.e., $R_{ub} = \sum_{i=1}^{N} R_i(u_i)$), which may be different from the optimal reward.

Lemma 1. *If the optimal reward is less than R_{ub}, then in any solution of 5.1-5.5, either the entire available slack is used (i.e., the processor is fully utilized) or all the tasks run at the minimum speed.*

Proof. By available slack we mean the CPU idle time. Using all the available slack in a schedule is equivalent to not having the CPU idle, that is, at any instance of time the processor is executing some task. Assume that there exists a solution **S** in which a task T_a runs on some speed $s_a > S_{min}$ and the entire available slack is not used. Since the optimal reward is less than R_{ub}, there also exists a task T_b that doesn't receive u_b cycles. We will show that a higher reward can be achieved for the same amount of energy, thus contradicting the fact that **S** is a solution of 5.1-5.5.

Using the available slack, the speed of a task T_k running at $s_k > S_{min}$ can be reduced to some $s'_k < s_k$ while still providing the same number of cycles to the task. If $s_b > S_{min}$, we pick task $k = b$ to reduce the speed. If $s_b = S_{min}$, we pick task $k = a$. If the slack allows it, then we can reduce s_k to $s'_k = S_{min}$, otherwise $S_{min} < s'_k < s_k$ and the processor is fully utilized. Either way, the total reward is not changed as we keep the number of cycles given to task T_k constant. However, due to the convex nature of the power functions, the total energy is reduced.

Next, the saved energy can be used to increase the speed of T_b while keeping t_b constant. Note that it is always possible to increase s_b, as s_b is known to be less than S_{max} by the way we picked T_k in the previous step. By keeping t_b constant and increasing s_b, the number of cycles allocated to task T_b will increase. Thus, the total reward increases, contradicting the maximum-reward condition. □

Lemma 2. *If the optimal reward is less than R_{ub}, then in any solution of 5.1-5.5, either the entire energy is used or all the tasks run at the maximum speed.*

Proof. Assume that there exists a solution **S** in which a task T_a runs on some speed $s_a < S_{max}$ and the entire available energy is not used. Also, since the optimal reward is less than R_{ub}, there exists a task T_b that does not receive u_b cycles. We will show how the extra energy can be used to increase the total reward, thus contradicting the fact that **S** is a solution of 5.1-5.5.

If $s_b < S_{max}$, we can use the extra energy to increase the speed s_b while keeping t_b unchanged and thus the total reward is increased. If $s_b = S_{max}$, we will improve the total reward as follows: first, using the available energy, the speed of T_a can be increased to some

$s'_a > s_a$ while still providing the same number of cycles C_a to the task. If the available energy allows it we can have $s'_a = S_{max}$, otherwise $S_{max} > s'_a > s_a$ and the entire energy is used. This will create some slack that can be used to slow down T_b while keeping C_b constant. Next, the energy saved by slowing down T_b can be used to increase the speed s_b and thus the total reward. Either way, the total reward is improved, contradicting the maximum-reward condition. □

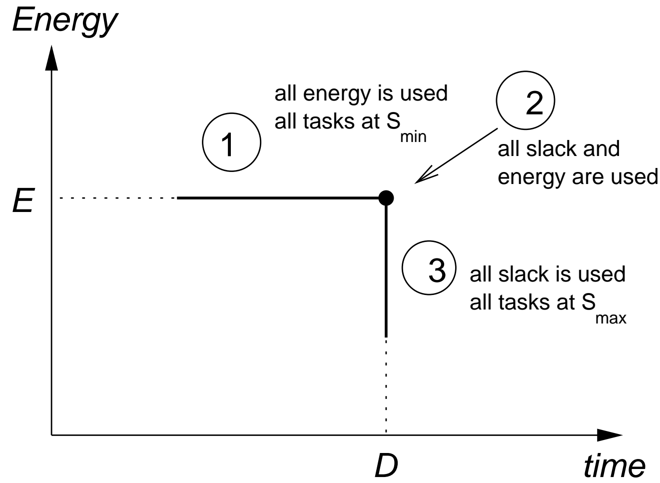


Figure 5.1: Design space

Theorem 1 below combines the results of the above two lemmas, showing that if the reward is less than R_{ub}, which is to be expected for most task sets, a solution *must* use the most amount of slack possible (unless all tasks are already running at S_{min}) and the most amount of energy possible (unless all tasks are running at S_{max}). In other words, the solutions are on the boundary of the design space, as seen in Figure 5.1.

Theorem 1. *If the optimal reward is less than R_{ub} then all solutions have one of the following properties: either (a) all the available slack and energy are used or (b) all the tasks are running at the same speed S_{min} or (c) all the tasks are running at the same speed S_{max}.*

Proof. If the optimal reward is less than R_{ub}, it follows from the above lemmas that there are only three possible types of solutions:

1. All tasks run at the minimum speed, in which case the entire available energy is used and there may be some slack in the final schedule.
2. All tasks run at the maximum speed, in which case the entire available slack is used but some energy may be wasted in the final schedule.
3. There is at least one task T_i running at speed $S_{min} < s_i < S_{max}$. In this case, it follows from Lemmas 1 and 2 that the entire slack and all the available energy are used.

□

Theorem 2 extends Theorem 1, removing the restriction on the reward, and showing that there always exists at least one solution that uses the most amount of slack and energy possible. This property is used to find the optimal solution for specific power functions and also in a proposed iterative algorithm for general power functions.

Theorem 2. *If the optimal reward is equal to R_{ub}, there exists a solution with one of the following properties: either (a) all the available slack and energy are used or (b) all the tasks are running at the same speed S_{min} or (c) all tasks are running at the same speed S_{max}.*

Proof. If the optimal reward is *exactly* R_{ub}, any solution can be transformed into a solution in which either all the available slack and energy are used or all the tasks run at the same speed S_{min} or S_{max}, as described next: If there is some available slack and a task T_i such that $s_i > S_{min}$, we transform the solution by slowing down T_i to use as much as possible of the available slack while keeping C_i constant. The transformation is applied repeatedly until either all tasks run at the minimum speed or all the available slack is used. Next, if there is some unused energy and a task T_j such that $s_j < S_{max}$, we transform the solution by speeding up T_j to use as much as possible of the extra energy while keeping t_j constant. The transformation is applied repeatedly until either all tasks run at maximum speed or all the available energy is used. □

The next theorem presents properties of the solutions that are used to identify the minimum-energy solution for specific power functions. Note that an implication of Theorem 1 is that if the optimal reward is less than R_{ub} then all solutions are also minimum-energy solutions. Theorem 3 shows how to compute the speed of all tasks if one task's speed

(different from S_{min} and S_{max}) is known. Knowing s_i, where $S_{min} < s_i < S_{max}$, the speed of all tasks T_j, $j \neq i$ can be determined as follows:

Theorem 3. *All minimum-energy solutions of 5.1-5.5 have the following properties:*

1. *If T_i and T_j are two tasks in the minimum-energy solution such that $S_{min} < s_i < S_{max}$ and $S_{min} < s_j < S_{max}$, then $P_i(s_i) - s_i \cdot P_i'(s_i) = P_j(s_j) - s_j \cdot P_j'(s_j)$.*
2. *If $s_i = S_{min}$ and $S_{min} < s_j < S_{max}$, then $P_i(s_i) - s_i \cdot P_i'(s_i) \leq P_j(s_j) - s_j \cdot P_j'(s_j)$.*
3. *If $s_i = S_{max}$ and $S_{min} < s_j < S_{max}$, then $P_i(s_i) - s_i \cdot P_i'(s_i) \geq P_j(s_j) - s_j \cdot P_j'(s_j)$.*

where P_i' and P_j' denote the first order derivatives of the power functions P_i and P_j.

Proof. Let **S** be a minimum-energy solution of 5.1-5.5 in which each task T_i receives C_i cycles. Since **S** is a minimum-energy solution, there is no other solution in which each task T_i receives exactly C_i cycles for less energy. Thus, **S** has to also be a solution of the following optimization problem:

$$\text{minimize} \quad \sum_{i=1}^{N} t_i \cdot P_i(s_i) \tag{5.11}$$

$$\text{subject to} \quad \sum_{i=1}^{N} t_i = D \tag{5.12}$$

$$t_i \cdot s_i = C_i \tag{5.13}$$

$$S_{min} \leq s_i \leq S_{max} \tag{5.14}$$

Using the Lagrangian multipliers method and Kuhn-Tucker conditions [56], any solution of 5.11-5.14 must satisfy the following:

$$P_i(s_i) + \lambda_0 + \lambda_i \cdot s_i = 0 \tag{5.15}$$

$$t_i \cdot P_i'(s_i) + \lambda_i \cdot t_i - \mu_i^1 + \mu_i^2 = 0 \tag{5.16}$$

$$\mu_i^1 \cdot (S_{min} - s_i) = 0 \tag{5.17}$$

$$\mu_i^2 \cdot (s_i - S_{max}) = 0 \tag{5.18}$$

where λ_0, λ_i, $\mu_i^1 \geq 0$, $\mu_i^2 \geq 0$, $i = 1, 2, \ldots, N$ are constants.

The three properties in the theorem are proved next.

1. Assume that T_i and T_j are two tasks in **S** such that $S_{min} < s_i < S_{max}$ and $S_{min} < s_j < S_{max}$. Then, from 5.17 and 5.18 it follows that $\mu_i^1 = \mu_i^2 = \mu_j^1 = \mu_j^2 = 0$. From 5.16 it results that $\lambda_i = -P_i'(s_i)$ and $\lambda_j = -P_j'(s_j)$. By substitution in 5.15, $P_i(s_i) - s_i \cdot P_i'(s_i) = P_j(s_j) - s_j \cdot P_j'(s_j)$.
2. Assume now that $s_i = S_{min}$ and $S_{min} < s_j < S_{max}$. Then, $\mu_i^2 = \mu_j^1 = \mu_j^2 = 0$. Since $\mu_i^1 \geq 0$, it results from 5.16 that $\lambda_i \geq -P_i'(s_i)$ and $\lambda_j = -P_j'(s_j)$. By substitution in 5.15, $P_i(s_i) - s_i \cdot P_i'(s_i) \leq P_j(s_j) - s_j \cdot P_j'(s_j)$.
3. Similarly, if $s_i = S_{max}$ and $S_{min} < s_j < S_{max}$, then $\mu_i^1 = \mu_j^1 = \mu_j^2 = 0$ and since $\mu_i^2 \geq 0$ it follows that $P_i(s_i) - s_i \cdot P_i'(s_i) \geq P_j(s_j) - s_j \cdot P_j'(s_j)$.

□

5.1.3 Optimal Solutions for Specific Power Functions

In [5] a polynomial time algorithm for optimally solving the following problem was given:

$$\text{maximize} \quad \sum_{i=1}^{N} f_i(t_i) \tag{5.19}$$

$$\text{subject to} \quad l_i \leq t_i \leq u_i \tag{5.20}$$

$$\sum_{i=1}^{N} t_i = D \tag{5.21}$$

Equations 5.19-5.21 correspond to the reward maximization problem without energy constraints. The proposed optimal algorithm, called OPT-LU, also assumes concave (or linear) functions f_i. The complexity of OPT-LU is $O(N^2 \log N)$. To use the OPT-LU technique as part of our solution to Equations 5.1-5.5, we have to replace the variables s_i with constants and eliminate Equation 5.5 from our model. This reduces the problem to a form that the OPT-LU algorithm can solve. To find s_i, we always look for solutions that consume all the available energy and slack, as seen in Theorem 2. The cases when all tasks run at the same speed S_{min} or S_{max} are treated as special cases.

5.1.3.1 Optimal solution for identical power functions

From Theorem 3 we can derive the following corollary:

Corollary 1. *If all tasks have identical power functions, then all tasks run at the same speed in the minimum-energy solution.*

Assuming identical power functions $P_i = P$, by virtue of Corollary 1 the total energy is minimized when all tasks run at the same speed s. In order to find the speed s that consumes all the energy in the given deadline we solve $P(s) = \frac{E_{max}}{D}$. Because of the speed bounds there are three possible cases:

1. $S_{min} \leq s \leq S_{max}$. In this case s is the optimal speed.
2. $s > S_{max}$. In this case there is plenty of energy to run all the tasks at the maximum speed. s is set to S_{max} and some of the energy has to be wasted.
3. $s < S_{min}$. The speed s is set to S_{min}. Observe that the entire slack cannot be used as the available energy would be exceeded even if all tasks run at S_{min}. Thus, the deadline is artificially reduced: $D' = \frac{E_{max}}{P(S_{min})}$.

After determining the speeds $s_i = s$ (and possibly adjusting the deadline), Equation 5.5 can safely be eliminated from the model, as the energy budget cannot be exceeded. Next, OPT-LU gives the optimal solution, if a solution exists.

5.1.3.2 Optimal solution for certain polynomial power functions

From Theorem 3 we can derive the following corollary:

Corollary 2. *For power functions of the type $P_i = \alpha_i \cdot s^q$, where q is constant for all tasks, the minimum-energy solutions have the following properties:*

1. *All tasks that don't run at minimum or maximum speed consume the same amount of power **P**.*
2. *All tasks running at S_{min} have a power consumption higher than **P**.*
3. *All tasks running at S_{max} have a power consumption less than **P**.*

Assuming no speed bounds ($S_{min} = 0$, $S_{max} = \infty$), it follows from Corollary 2 that in the minimum-energy solution all tasks run at the same power P. Thus, the CPU power that

fully utilizes the available energy in the given deadline is $P = \frac{E_{max}}{D}$. Next, the speeds s_i of all tasks T_i are determined from the equations $P_i(s_i) = P$. Having determined all speeds, Equation 5.5 is eliminated from the model and the problem is transformed into an OPT-LU problem which returns the optimal solution, if a solution exists.

The solution returned is also minimum-energy if the total reward is less than R_{ub}. If the reward is exactly R_{ub}, then, analogous to Section 5.1.3.1, the minimum-energy speed and execution time for task T_i are $s_i' = s_i \frac{\sum_{i=1}^{N} t_i}{D}$ and $t_i' = t_i \frac{D}{\sum_{i=1}^{N} t_i}$, respectively.

5.1.4 Iterative DVS Algorithm for General Power Functions

We next propose an iterative algorithm for general power functions. Similar to the optimal solutions in Sections 5.1.3.1 and 5.1.3.2, the algorithm for the general case starts by using the OPT-LU algorithm [5] to first solve the problem without energy constraints. Clearly, the optimal reward assuming limited energy E is bounded by the OPT-LU solution assuming all the tasks are running at speed S_{max}. The unlimited-energy solution is then refined to satisfy the energy constraint by three types of *transfers*, as described next.

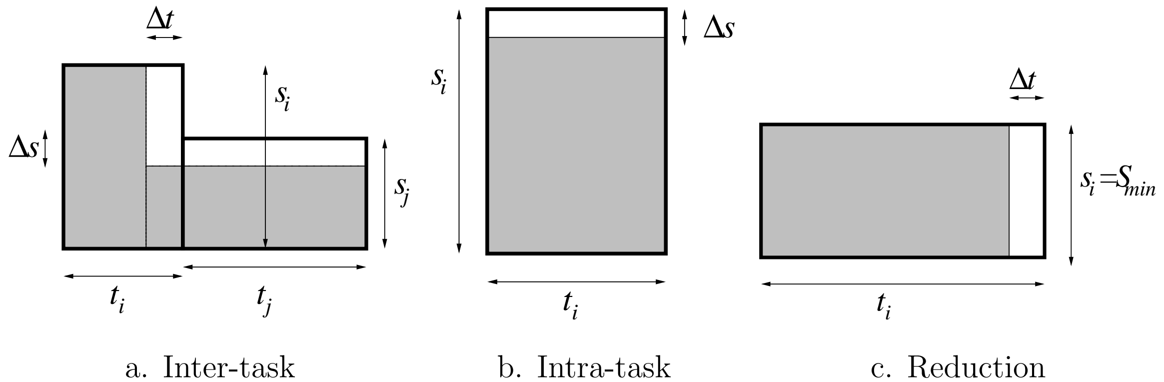


Figure 5.2: Transfer types

5.1.4.1 Inter-Task Transfers Based on Theorems 1 and 2 the algorithm looks for solutions that use all the available energy and slack. The algorithm iteratively transfers Δt units of time from task T_i to task T_j, $i \neq j$, by keeping the speed of T_i the same (and thus decreasing the number of cycles and reward of T_i) and reducing the speed of T_j by $\Delta s = \frac{s_j \Delta t}{t_j + \Delta t}$ so as

to preserve the same number of cycles C_j. This type of transfer is referred to as *inter-task* transfer, and is illustrated in Figure 5.2a. The rectangles represent tasks expressed by speed (the height of the rectangle in cycles per time unit) and execution time (the width in time units). Hence, the area of a task represents the demand of the task in terms of number of cycles. The bold rectangles show the tasks before the transfer, the shadowed rectangles show the tasks after the transfer.

5.1.4.2 Intra-Task Transfers Similarly, in a *intra-task* transfer, a task T_i can transfer Δt time units to itself thus reducing its speed by $\Delta s = s_i \frac{\Delta t}{t_i}$, and thus reducing its reward. Intra-task transfers are shown in Figure 5.2b. The intra-task transfer results in a decrease in the energy consumption at the expense of a decrease in the reward. Intra and inter-task transfers stop when the total energy of the current schedule satisfies the budget constraint.

5.1.4.3 Reduction If no transfer is possible and the energy constraint is not satisfied, it is the case that all tasks run on the minimum speed and the entire available slack is not used. For this situation a *reduction* is introduced: reduce the execution time of task T_i by Δt, as illustrated in Figure 5.2c.

5.1.4.4 Algorithm Algorithm 1 carries out the transfers and reductions. The input parameters are the unlimited energy solution $\mathbf{S}$=$\{(S_{max}, t_1), (S_{max}, t_2), \ldots, (S_{max}, t_n)\}$ obtained using the OPT-LU algorithm and the value of the interval Δt. $E(\mathbf{S})$ denotes the total energy of an intermediate solution $\mathbf{S}$. Inter-task and intra-task transfers are in lines 2-4, reductions are in lines 6-9.

Clearly, the value of Δt will directly influence the quality and the running time of the solution. When Δt tends to zero, the solution tends to the optimum and the running time increases. As the experiments will show the error obtained also depends on other factors such as the amount of energy available. The Δt size that gives an acceptable solution cannot be determined a priori. Therefore, we propose an algorithm that starts with an initial $\Delta t = \Delta t_{initial}$ and iteratively refines the solution by halving Δt until a stopping criterion

Algorithm 1 SOLVE($\Delta t, \mathbf{S}$) algorithm

1: **while** $E(\mathbf{S}) > E$ **do**
2: determine transferring task T_i and receiving task T_j
3: **if** T_i and T_j exist **then**
4: execute transfer, update $\mathbf{S}$, $E(\mathbf{S})$
5: **else**
6: **while** $E(\mathbf{S}) > E$ **do**
7: determine task T_i from which to take Δt time units
8: **if** T_i exists **then**
9: execute reduction, update $\mathbf{S}$, $E(\mathbf{S})$
10: **else**
11: **return** failure ($\mathbf{S}$=NULL,$\mathbf{R}$=0)
12: **end if**
13: **end while**
14: **end if**
15: **end while**
16: **return** $\mathbf{S}$=$\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$, and $\mathbf{R}$= $\sum_{i=1}^{N} R_i(t_i \cdot s_i)$

suggests that the quality of the solution is acceptable.

The stopping criterion used for the quality of the solution is: $\frac{R_{\Delta t} - R_{2\Delta t}}{R_{\Delta t}} \leq \varepsilon$, where $R_{\Delta t}$ and $R_{2\Delta t}$ are the rewards for the current and previous iteration, and ε is a given threshold. To prevent searching indefinitely for a solution we use another stopping criterion, Δt_{min}, as a lower bound for Δt (that is, the search stops when $\Delta t < \Delta t_{min}$). The algorithm is presented in Figure 2.

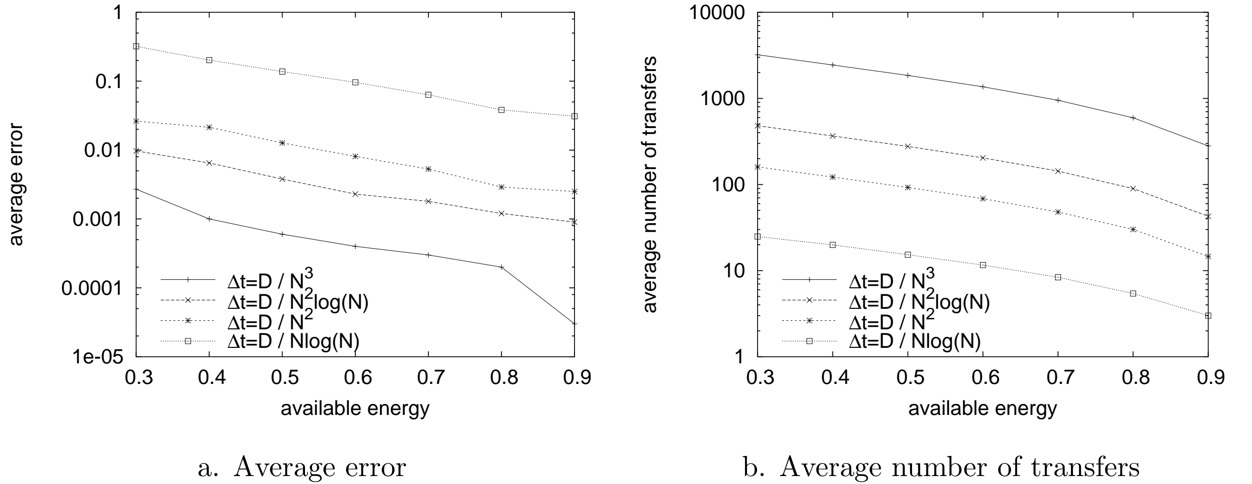
Algorithm 2 Variable Δt algorithm

1: initialize $\Delta t = \Delta t_{initial}$ and $R_{\Delta t} = 0$
2: **while true do**
3: $R_{2\Delta t} = R_{\Delta t}$
4: **if** $\Delta t < \Delta t_{min}$ **then**
5: **return** failure
6: **end if**
7: SOLVE($\Delta t, \mathbf{S}$) returns $R_{\Delta t}$ and $S_{\Delta t}$
8: **if** $R_{\Delta t} \neq 0$ and $\frac{R_{\Delta t} - R_{2\Delta t}}{R_{\Delta t}} \leq \varepsilon$ **then**
9: **break**
10: **end if**
11: $\Delta t = \frac{\Delta t}{2}$
12: **end while**
13: **return** $S_{\Delta t}$, $R_{\Delta t}$

5.1.5 Evaluation

We first validate the SOLVE($\Delta t, \mathbf{S}$) algorithm. We simulate task sets with $N = 20$ tasks and identical power functions of the type $P = s_i^q$, where for each simulation run, q was set to 2 or 3 (square and cubic power functions). We use linear reward functions of the type $R_i(t_i \cdot s_i) = \beta_i \cdot t_i \cdot s_i$, where the coefficients β_i were randomly chosen such that $\beta_i \in [0, N]$. The lower computation bounds l_i and u_i were randomly generated so that $l_i \in [1, N]$ and $u_i \in [l_i, l_i + N]$. The deadline D was generated in the range $[\sum_{i=1}^{N} \frac{l_i}{S_{max}}, \sum_{i=1}^{N} \frac{u_i}{S_{max}}]$. After solving the energy unrestricted OPT-LU, E_{max} was chosen so that it does not exceed the energy of the OPT-LU solution: $E_{max} \in [\frac{1}{5} E_{OPT-LU}, E_{OPT-LU}]$. For the speed limits we used the normalized speeds $S_{max} = 1$ and $S_{min} = 0.5$. We note that similar results are obtained with narrower ranges for β_i, l_i and u_i, as well as $S_{min} < 0.5$. We simulated 1000 task sets for each point in the graphs.

The error of a solution $\mathbf{S}$ is defined as $e = \frac{R_{opt} - R(\mathbf{S})}{R_{opt}}$, where R_{opt} is the total reward of the optimal solution and $R(\mathbf{S})$ is the total reward of solution $\mathbf{S}$. Note that R_{opt} can always be computed for identical power functions, as shown in Section 5.1.3.1.



a. Average error

b. Average number of transfers

Figure 5.3: Fixed-size transfers, identical power functions

The effect of the value of Δt on the accuracy of the solution, is shown in Figure 5.3, for a wide range of Δt values. For example, a value of $\Delta t = \frac{D}{N \log N}$ is coarse-grained, as each

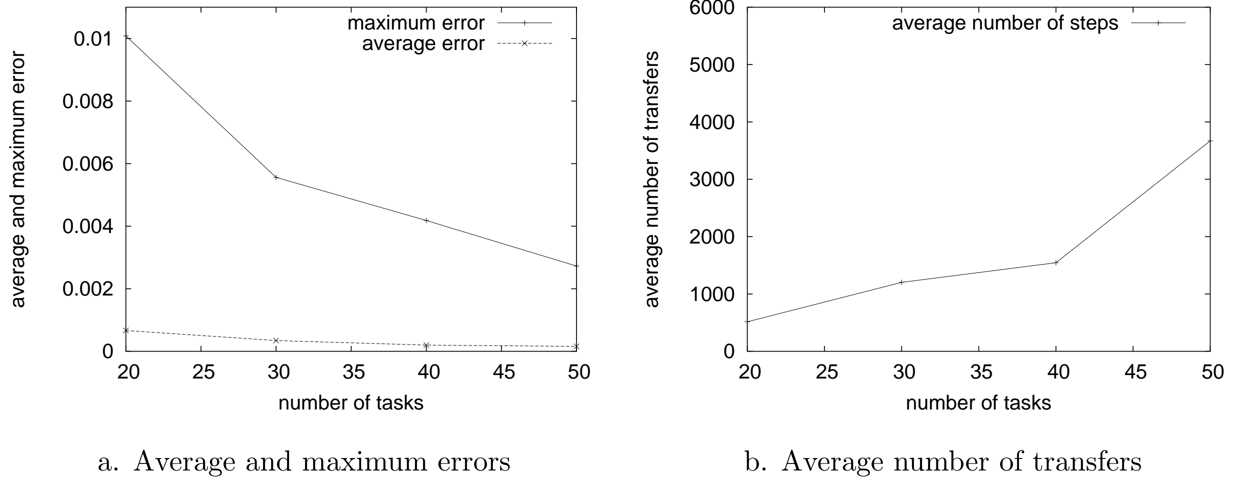
task has on average only 3 ($\log N$, for $N = 20$ tasks) Δt intervals that can be transfered, while a value of $\Delta t = \frac{D}{N^3}$ is fine-grained, with 400 intervals on average for each task. As seen in the figure, a fixed value of $\Delta t = \frac{D}{N^2 \log N}$ results in less than 1% error, with a relatively small number of transfers. However, if a better accuracy is desired or if the energy available is relatively small, $\Delta t = \frac{D}{N^2 \log N}$ might not be enough. On the other hand, for most task sets a higher Δt (fewer transfers) can achieve a satisfactory solution. The second algorithm (variable Δt size) was proposed to handle these problems.

We used the variable Δt version of our algorithm and compared the solution we obtained with the optimal solution. For this experiment we use different power functions of the type $P_i = \alpha_i \cdot s_i^q$, where q is constant for all tasks (for identical power functions we obtained quite similar results). As shown in Section 5.1.3.2, the optimal solution can always be computed for power functions of the type $P_i = \alpha_i \cdot s_i^q$ if there are no speed bounds. Thus, the power coefficients were generated so that in the optimal solution no task runs on the minimum or maximum speed. We ensured this by first generating the speeds s_i in the range $[S_{min}, S_{max}] = [0.5, 1]$ and the CPU power P at which all tasks run in the optimal solution. Only after that the coefficients α_i were generated such that $\alpha_i \cdot s_i^q = P$. We use concave reward functions of the type $R_i(C_i) = \ln(\beta_i \cdot C_i + 1)$ with random coefficients $\beta_i \in [0, N]$. As seen in Figure 5.4, with a stopping criterion of $\varepsilon = 0.0005$ and task sets of up to 50 tasks, the average error is less than 0.1%.

Observe that for the first two experiments it was possible to compare the algorithm with the optimal solution. Since we did not prove that the algorithm converges to the optimal, we consider these experiments necessary to evaluate the performance of the algorithm. A third experiment is intended to be more realistic and assumes power functions extracted from real processor models [48]. In particular, the power functions used are of the type:

$$P_i = \alpha_i \cdot [0.248s^3 + 0.225s^2 + \sqrt{(311s^2 + 282s) \cdot (0.014s^2 + 0.0064s)}] \tag{5.22}$$

However, we have no way of computing the optimal solution in this case, and hence, we approximate the optimal with the solution obtained for a very small Δt (we used $\Delta t = \frac{D}{10^6}$) and report average and maximum errors relative to this solution. For comparison, we also



a. Average and maximum errors

b. Average number of transfers

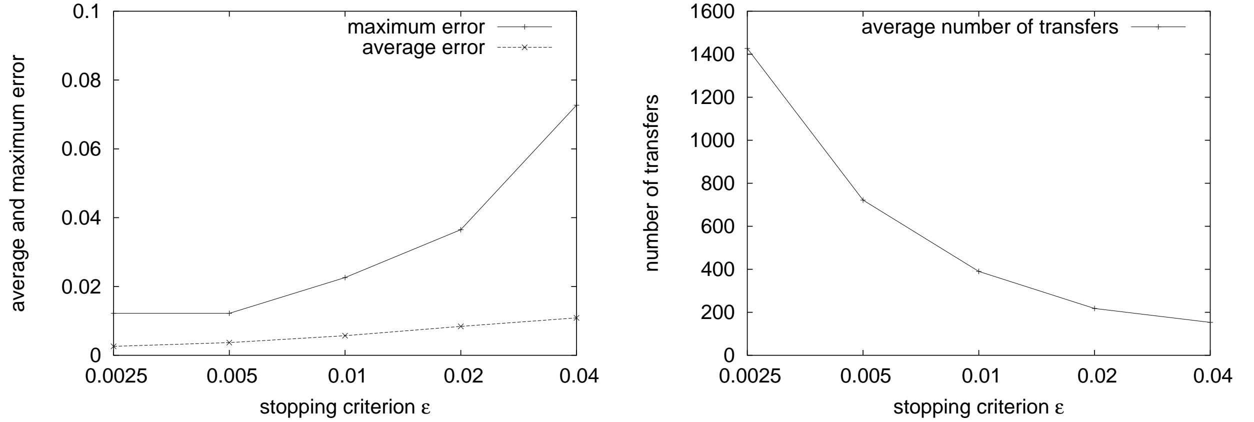
Figure 5.4: Variable-size transfers, polynomial power functions

used $\Delta t = \frac{D}{10^6}$ in the previous experiment for different power functions of the type $P_i = \alpha_i \cdot s^q$ and the maximum absolute error (exactly computed) across all simulations was less than 0.0001.

The running time of the algorithm depends on the value of ε and the number of tasks N. For $\varepsilon = 0.01$ and $N = 50$ tasks the running time is in the range of milliseconds (average of 38 milliseconds on a 400MHz Pentium II), with just 1.5 milliseconds for $N = 20$ tasks and half a millisecond for $N = 20$ tasks and simpler power functions (such as $P_i = \alpha_i \cdot s^q$). The average errors and number of transfers as a function of ε are shown in Figure 5.5. Results are very similar, with less than 1% average error. As well, we note the number of halvings is very small. For example, for $\Delta t_{initial} = \frac{D}{N \log N}$ only two halvings are required on average for $\varepsilon = 0.01$.

5.2 DISCRETE REWARDS AND POWER FUNCTIONS

This section describes the case of discrete power and reward functions. Unlike the continuous case, only a few (discrete) processor frequencies are available for dynamic voltage scaling (DVS), corresponding to the realistic case of processors that support DVS (the XScale pro-



a. Average and maximum errors

b. Average number of transfers

Figure 5.5: Variable-size transfers, realistic power functions

cessor model [57] with 5 frequencies/voltages is used for evaluation purposes). The discrete reward model corresponds to applications that do not reward partial execution. Task versions are proposed as an extension of the reward model corresponding to step reward functions (Figure 3.1b).

We first propose the *REW-Pack* and *REW-Unpack* algorithms for single-version task sets. For tasks with multiple versions, the *MV-Pack* algorithm is proposed. Note that, as in the continuous case, the frame-based and periodic task models result in equivalent problem definitions [70].

5.2.1 Optional Single Version

Using the notation introduced in Chapter 3, the problem formulation for the discrete all-or-nothing reward model is presented next. The problem is similar to the continuous case, with two exceptions: tasks are optional and non-adaptive. Thus, a task is either selected for execution, in which case its reward is added to the system reward, or is omitted and contributes no reward to the system. Formally, the problem is to determine the subset of tasks S selected for execution and their speeds s_i so that to:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in S} r_i && (5.23) \\ \text{subject to} \quad & \sum_{i \in S} t_{i,s_i} \leq D && (5.24) \\ & \sum_{i \in S} e_{i,s_i} \leq E && (5.25) \\ & S \subseteq \{1, 2, \ldots, N\} && (5.26) \\ & s_i \in \{1, 2, \ldots, M\} && (5.27) \end{aligned}$$

The maximization objective is present in Equation 5.23. Inequality 5.24 represents the real-time constraint for frame-based systems, and inequality 5.25 enforces the energy budget. The problem was shown to be NP-hard in [68].

5.2.1.1 The REW-Pack Algorithm We propose the *REW-Pack* algorithm to solve the problem described by Equations 5.23-5.27, for tasks that have a single version. Each task is optional (i.e., may be omitted in the final solution).

As with the continuous reward and power models, we look for solutions that use all available time and energy. The iterative algorithm for continuous rewards starts with an optimal solution assuming unlimited energy and relies on fine-grained schedule adaptation to satisfy the energy constraint (Figure 5.2). However, in the discrete case, identifying the optimal solution is NP-hard even for unlimited energy, and schedule adaptation is coarse-grained, due to the all-or-nothing reward model. While the iterative algorithm performs three types of fine-grained intra-task or inter-task transfers, *REW-Pack* relies on three types of coarse-grained schedule adaptations: adding a task, increasing the speed of a task, and dropping a task. Starting with a low-energy schedule, the search stops when the energy constraint is violated.

The flowchart of *REW-Pack* is presented in Figure 5.6. The algorithm works as follows: a new task is scheduled (at the minimum speed) as long as the energy constraint is not violated. When the deadline constraint is no longer satisfied, the algorithm starts *packing* already selected tasks to create slack for the remaining tasks, where packing means to decrease the running time of a task, by increasing the speed of the selected task.

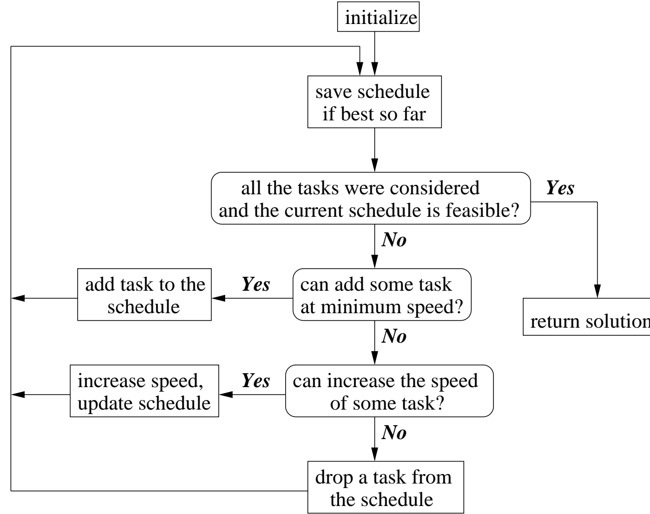


Figure 5.6: Flowchart of *REW-Pack*

The complete *REW-Pack* algorithm is shown in Algorithm 3. *add_task()*, *drop_task()* and *increase_speed()* all return the task number or -1 if no task can be chosen. The variables *time* and *energy* store the total running time of the schedule ($time = \sum_{i \in S} t_{i,s_i}$) and the total energy consumed ($energy = \sum_{i \in S} e_{i,s_i}$) and are initialized to zero. V stores the system value for the current schedule ($V = \sum_{i \in S} r_i$) and SV stores the *system value*, that is, the largest value of V encountered thus far.

The task T_i that is added (always at the minimum speed) to the current schedule must satisfy all of the following criteria: the task was not considered yet ($considered[i] = false$), the current schedule is feasible ($time \leq D$) and by adding the task to the current schedule at the minimum speed the energy budget is not exceeded ($energy + e_{i,1} \leq E$). Among all the tasks that satisfy the above criteria, we select the one that has the largest ratio $\frac{r_i}{t_{i,1} e_{i,1}}$. Thus, a task is a good candidate if it has large reward, small running time and/or small energy consumption. In our experiments, metrics that do not consider all parameters (i.e., task value, task energy and task time) failed to give good approximations of the optimal solution.

If no task can be added to the schedule, the algorithm packs tasks to make room for other not yet selected tasks, where packing means to increase the speed of one of the selected tasks,

Algorithm 3 The *REW-Pack* algorithm

```
1: Initialize: selected[i]=false, considered[i]=false ∀i
2: energy = 0, time = 0, SV = 0, V = 0
3: while true do
4:   if time ≤ D and SV < V then
5:     SV = V, sol_selected[i]=selected[i], sol_speed[i]=s_i, ∀i
6:   end if
7:   if considered[i]==true ∀i and time ≤ D then
8:     break
9:   end if
10:  i=add_task()
11:  if i≠-1 then
12:    selected[i]=true, considered[i]=true
13:    s_i = 1, energy+ = e_{i,1}, time+ = t_{i,1}, V+ = r_i
14:  else
15:    i=increase_speed()
16:    if i≠-1 then
17:      energy+ = e_{i,s_i+1} − e_{i,s_i}
18:      time+ = t_{i,s_i+1} − t_{i,s_i}, s_i = s_i + 1
19:    else
20:      i=drop_task()
21:      energy− = e_{i,s_i}, time− = t_{i,s_i}
22:      V− = r_i, selected[i]=false
23:    end if
24:  end if
25: end while
26: return solution (sol_selected, sol_speed, SV)
```

always to the next higher speed level. The task chosen for a speed increase must satisfy the following: it must be selected in the current schedule ($selected[i] = true$), it is not running at the maximum speed ($s_i \neq M$) and by increasing its speed to the next higher speed level the energy budget is not exceeded ($energy + e_{i,s_i+1} - e_{i,s_i} \leq E$). Among all tasks that satisfy the above criteria, we select (for packing) the task T_i with the highest ratio $\frac{\Delta t}{\Delta E}$, where $\Delta t = t_{i,s_i} - t_{i,s_i+1}$ and $\Delta E = e_{i,s_i+1} - e_{i,s_i}$. Thus, the best candidates for packing are considered the tasks that create a lot of room (time or slack) for the remaining tasks while not significantly increasing the energy consumption.

If the previous two steps fail, a task is eliminated from the current schedule. The task that is dropped is the task in the current schedule ($selected[i] = true$) which has the smallest ratio $\frac{r_i}{t_{i,s_i} e_{i,s_i}}$. Once a task is dropped, it is not added again.

The complexity of the *REW-Pack* algorithm can be analyzed as follows. Each task is

added at most once and dropped at most once. For each task, its speed can be increased at most $M-1$ times. Determining what task to pick takes $\log N$ time for all functions (add, increase and drop). Thus, the complexity of the algorithm is $O(MN \log N)$.

5.2.1.2 The REW-Unpack Algorithm With the *REW-Unpack* algorithm, the search goes in the opposite direction: tasks are added at the maximum speed and the schedule is *unpacked* (i.e., a task is selected and its speed decreased) to create energy for the remaining tasks. Thus, while *REW-Pack* starts with a low-energy schedule and does not allow exceeding the energy budget, *REW-Unpack* starts with a high-energy schedule and does not allow exceeding the time budget. The function *increase_speed*() is replaced with *decrease_speed*(). The same metrics and complexity of *REW-Pack* apply.

5.2.1.3 Evaluation One key desired feature of the algorithms is fast running times. This allows a scheduler to adapt dynamically to changes in the system such as tasks becoming unavailable, new tasks being added to the system or new timing and energy constraints. Simulations results on the XScale processor power model (shown in Table 5.1) show that both algorithms return solutions very close to the optimal (within 3%) with running times in the microseconds range (on a 850MHz Pentium III with 256MB of RAM), even for hundreds of tasks.

Table 5.1: Frequency/voltage settings for Intel XScale processors

$f(MHz)$	1000	800	600	400	150
$V_{dd}(V)$	1.80	1.60	1.30	1.00	0.75

We simulated both algorithms on the same task sets and, for relatively small task sets, compared our solution with the optimal solution, obtained through an exhaustive search. We define the absolute error for any of the two algorithms to be $\frac{SV_{OPT}-SV}{SV_{OPT}}$, where SV represents the system value (reward) resulting from the algorithm and SV_{OPT} is the optimal system value. The average error for several experiments is defined as the arithmetic mean of the absolute errors for each experiment.

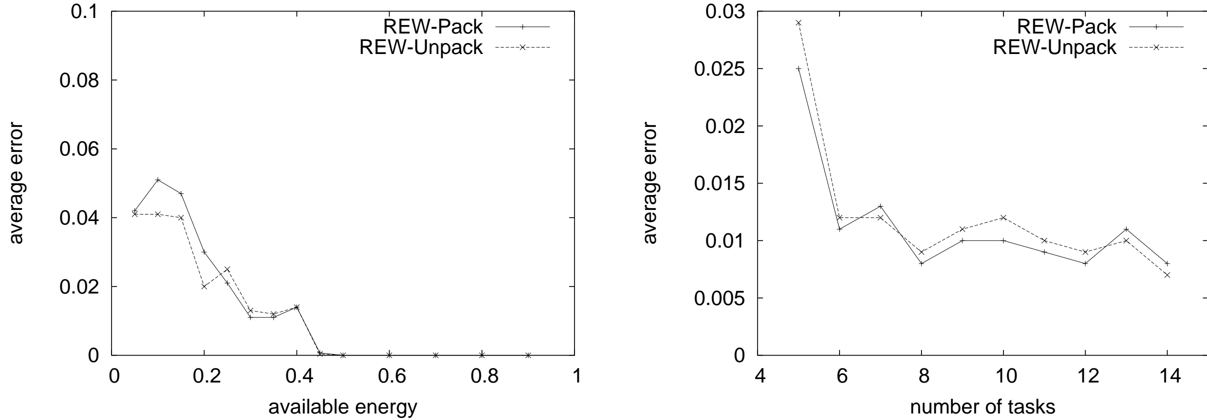
The simulations are described by the following parameters: (1) N - number of tasks (2) M - number of speed levels (3) $t_{i,j}$, $e_{i,j}$ - time and energy requirements (4) D - deadline (5) E - available energy and (6) r_i - task values.

For each task, its execution time at minimum speed $t_{i,1}$ was randomly generated in the range $[1, 100]$, resulting in a wide variation of task execution times. The running time of task T_i at speed level j was then computed as $t_{i,j} = t_{i,1}\frac{f_1}{f_j}$, thus the running time is inversely proportional to the speed. Inspired from Equation 3.1, the power consumption of task T_i at speed level j, is given by the formula $P_{ij} = a_i \cdot Voltage(j)^2 \frac{f_j}{f_M}$. Thus, the power is proportional to the normalized speed and the square of the voltage. a_i is an activity factor different for each task, proportional to the dynamic switching caused by the task and randomly generated in the range $[0.8, 1.2]$. The energy requirement $e_{i,j}$ is then computed as $e_{i,j} = P_{i,j} \cdot t_{i,j}$, that is the power multiplied with the time. For a wide variation of task rewards, the value r_i was generated randomly in the range $[1, 100]$ for each task.

The maximum deadline, Max_D, is defined as $Max_D = \sum_{i=1}^{N} t_{i,1}$, that is the total execution time of the tasks at minimum speed. The maximum energy, Max_E, is defined as $Max_E = \sum_{i=1}^{N} e_{i,M}$, that is the total energy requirement for all tasks if running at the maximum speed. Clearly, if $D \geq Max_D$ the timing constraint cannot be violated. Similarly, if $E_{max} \geq Max_E$ the available energy cannot be exceeded. Two parameters: α and β describe the available time and energy in the system. The deadline was generated using the formula $D = \alpha \cdot Max_D$ and the energy was generated by $E_{max} = \beta \cdot Max_E$, where $\alpha \in [0, 1]$ and $\beta \in [0, 1]$ (thus covering the full range of time and energy values).

Figure 5.7a shows the average error for tight timing constraints ($\alpha = 0.2$) as a function of the available energy (β) for $N = 10$ tasks. The average error quickly tends to zero as more energy is available.

The exponential nature of the optimal solution makes it hard to compute the absolute error for large values of N. There is experimental evidence, however, that the absolute errors do not increase (rather, they actually decrease) as the number of tasks increases. For example, in Figure 5.7b, where we simulated task sets with 5 to 14 tasks for $\alpha = 0.3$ and $\beta = 0.3$, we can see this trend.



a. Average error as a function of energy (β) b. Average error as a function of N

Figure 5.7: Evaluation of *REW-Pack* and *REW-Unpack*

As seen in Figure 5.7, *REW-Pack* and *REW-Unpack* have very similar average errors. Intuitively *REW-Pack* should perform better on time-constrained task sets and *REW-Unpack* should have better results on energy-constrained task sets. It turns out however that the time and energy are equally important (except for cases when D or E are too large to be used entirely given the other constraint) and both algorithms return schedules that use on average more than 90% of both the available time and energy. A graphical demonstration of the two heuristics is available at `http://www.cs.pitt.edu/PARTS/demos`.

5.2.2 Multiple Task Versions

The discrete reward model in Equations 5.23-5.27 corresponds to a single version for each task, which is either selected or omitted in the final schedule. To allow for better application adaptation, we propose to enhance the discrete reward model with task versions. In this model, each task may have several versions, each with different rewards and resource requirements (energy and time), and no reward for partial execution. For example, one version may require less cycles and therefore use less energy, at the expense of producing less accurate/complete/valuable results.

Equivalently, task versions also correspond to the same application with discrete completion points, as described in Section 3.2.2. This effectively results in the discretized version

of the IC reward model. Unfortunately, although the three types of transfers for IC tasks (Section 5.1.4) can be adapted for discrete rewards, the iterative algorithm for continuous rewards still cannot apply to the task versions model. This is because it depends on the optimal solution assuming unlimited energy, which is NP-hard for the case of step reward functions. Thus, we propose a new algorithm, *MV-Pack*, for the case of multiple task versions.

Note that all tasks must be selected in the solution. However, the *MV-Pack* algorithm can also handle a combination of optional and mandatory tasks. For this case, the original task set is modified in the following way: for each optional task we artificially add a version with zero reward and zero energy and time requirements. We call this added version the *zero version*. A task selected in the final schedule at its zero version is equivalent to a task not selected for execution.

Formally, the problem is to determine for each task T_i its speed level s_i, as well as its version v_i, so that to:

$$\text{maximize} \quad \sum_{i=1}^{N} r_i^{v_i} \tag{5.28}$$

$$\text{subject to} \quad \sum_{i=1}^{N} t_{i,s_i}^{v_i} \leq D \tag{5.29}$$

$$\sum_{i=1}^{N} e_{i,s_i}^{v_i} \leq E \tag{5.30}$$

$$v_i \in \{1, 2, \ldots, V\} \tag{5.31}$$

$$s_i \in \{1, 2, \ldots, M\} \tag{5.32}$$

The problem was shown to be NP-hard in [70]. Note that all tasks must be selected in the final schedule, unlike the single-version problem definition. This restriction is eliminated in the next section, allowing tasks to be either mandatory or optional.

5.2.2.1 The MV-Pack Algorithm The *MV-Pack* algorithm is an extension of *REW-Pack* for multiple-version task sets. The flowchart of *MV-Pack* is shown in Figure 5.8. The algorithm has three major components: add task, increase speed and increase version. The first two components are identical with those of *REW-Pack*. However, since the multiple

versions task model requires that each task is selected in the schedule, tasks are never dropped.

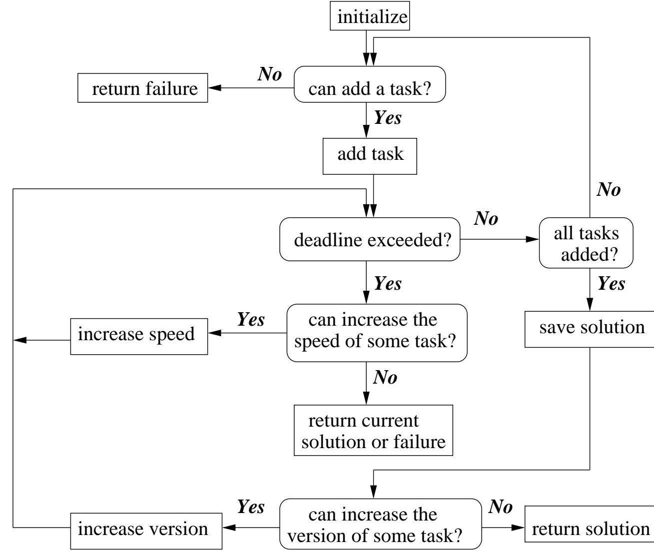


Figure 5.8: Flowchart of *MV-Pack*

The algorithm starts with an empty schedule. A new task is added if possible, always at the first (smallest) speed level and version (we assume that task versions are sorted by their reward - the first version has the smallest reward). If the deadline is exceeded, tasks are packed to make room for other tasks. When all the tasks are selected in the schedule, a minimum reward solution is found, otherwise failure is returned.

Next, while the remaining energy allows it, a better schedule (higher reward) is searched by increasing the version of some task. The third component of the algorithm (increase version) selects the task to move to its next higher version. The old version is removed from the schedule, while the new version is added at the minimum speed. Tasks are then packed if necessary until either a solution with the new version is found or the energy is exceeded, in which case the current solution is returned.

The task T_i that is selected to move to the next higher reward version satisfies the following: (1) it is not running at the highest version ($v_i < V$) (2) by replacing the current version with the next higher version at the first speed level, the energy budget is not exceeded ($energy + e_{i,1}^{v_i+1} - e_{i,s_i}^{v_i} \leq E$) and (3) among all the tasks that are not running at their highest

version, the next version at minimum speed has the largest reward per unit time and energy. That is, we select task i that maximizes $\frac{r_i^{v_i+1}}{t_{i,1}^{v_i+1} \cdot e_{i,1}^{v_i+1}}$.

The complexity of *MV-Pack* can be analyzed as follows. Each task is added at most once and its version can be increased at most $V-1$ times. For each task we can increase its speed at most $(M-1) \cdot V$ times. With appropriate data structures, determining which task to choose takes $\log N$ time for all functions (add task, increase speed and increase version). Thus, the complexity of the algorithm is $O(MVN \log N)$.

5.2.2.2 Evaluation We simulated the Intel XScale architecture (Table 5.1) for task sets with $V = 4$ versions. For each task, the execution time of the first version at minimum speed $t_{i,1}^1$ was randomly generated in the range $[10, 100]$. For the remaining versions, the running time at the first speed level was generated by the formula $t_{i,1}^k = t_{i,1}^{k-1} + \Delta_i^k$, where $\Delta_i^k \in [0.2 \cdot t_{i,1}^1, 1.2 \cdot t_{i,1}^1]$ was randomly generated for each task version. Next, $t_{i,j}^k$ was computed for all versions and all speed levels, inversely proportional to the speed ($t_{i,j}^k = t_{i,1}^k \cdot \frac{f_1}{f_j}$).

For the power consumption of a task version T_i^k at speed level j, we use the formula $P_{i,j}^k = a_i \cdot Voltage(j)^2 \frac{f_j}{f_M}$. Thus, the power is proportional to the normalized speed and the square of the voltage. a_i is an activity factor different for each task and identical for all versions of the same task, proportional to the dynamic switching caused by the task and randomly generated in the range $[0.8, 1.2]$. The energy requirement $e_{i,j}^k$ is then computed as $e_{i,j}^k = P_{i,j}^k \cdot t_{i,j}^k$, that is the power multiplied with the time.

Task values of the first versions r_i^1 were generated randomly in the range $[10, 100]$. For the higher versions, task rewards were generated according to the formula $r_i^k = r_i^{k-1} + \delta_i^k$, where $\delta_i^k \in [0.2 \cdot r_i^1, 1.2 \cdot r_i^1]$ was randomly generated for each task version. Thus, observe that each version requires more time and more energy than the previous versions, but gives a higher reward; also, there is no assumption on the shape of the reward function (i.e., it is not necessarily convex, linear or concave).

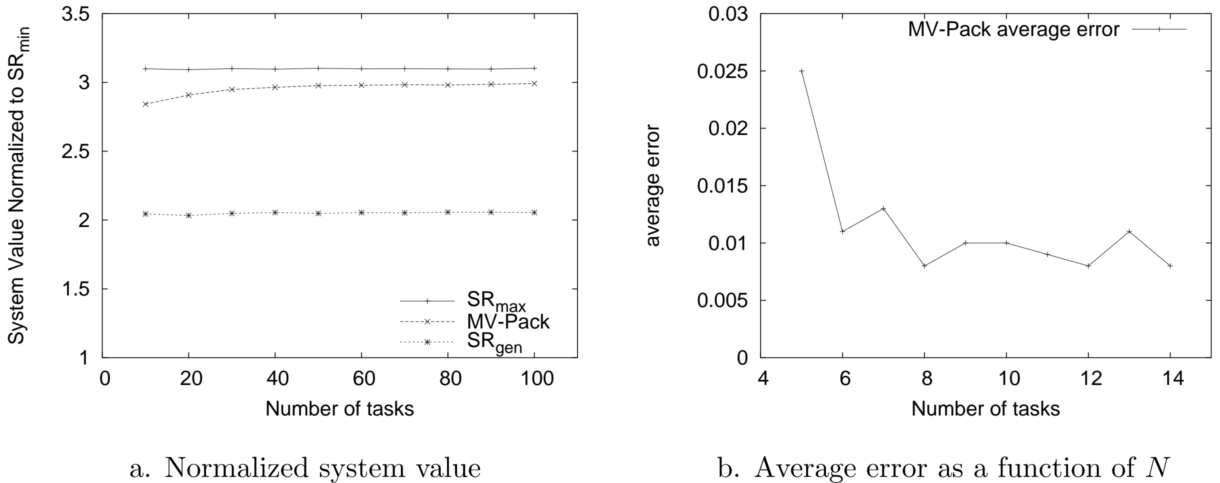
Experiments with different ranges for δ_i^k and Δ_i^k (such as $[10, 100]$), also with narrower or broader ranges for the activity factors a_i (such as $[0.2, 1.2]$) produced very similar results.

The deadline D and maximum energy E are generated by the formulas $D = \sum_{i=1}^{N} t_{i,s_i}^{v_i}$ and respectively $E = \sum_{i=1}^{N} e_{i,s_i}^{v_i}$, where the indexes for speed $s_i \in \{1, 2, \ldots, M\}$ and for value

$v_i \in \{1, 2, \ldots, V\}$ are randomly picked for each task $i \in \{1, 2, \ldots, N\}$.

We denote by SR_{min} the minimum system reward that can be achieved for a given task set, $SR_{min} = \sum_{i=1}^{N} r_i^1$. Similarly, SR_{max} denotes the maximum reward that can be achieved, $SR_{max} = \sum_{i=1}^{N} r_i^V$. Observe that if each task i runs at the version v_i and the speed level s_i used to generate D and E, the energy and deadlines are not exceeded and the system reward is $SR_{gen} = \sum_{i=1}^{N} r_i^{v_i}$.

Since it is impractical to compute the optimal solution, we will compare the performance of *MV-Pack* with SR_{min}, SR_{max} and SR_{gen}. Figure 5.9a shows the comparison for task sets of 10 to 100 tasks, where SR_{gen}, SR_{max} and the reward returned by the algorithm are normalized to SR_{min}. Each point is the average of 1000 simulation runs. In all experiments, *MV-Pack* returned a system value higher than SR_{gen} and close to SR_{max}. Note that SR_{max} is an upper bound on the optimal solution, not the optimal solution itself. In most cases SR_{max} cannot be achieved without exceeding the deadline or energy restrictions. For most graph points *MV-Pack* used more than 99% of the available energy; the smallest value is 94%. Similarly, *MV-Pack* used at least 97% of the available time. The average error as a function of N for single-version tasks is shown in Figure 5.9b. The actual running times of the algorithm is less than a millisecond for all experiments, on a 850 MHz Pentium III.



a. Normalized system value

b. Average error as a function of N

Figure 5.9: Evaluation of *MV-Pack*

5.3 LONG-TERM REWARD MAXIMIZATION

While the previous algorithms assume a given energy budget for a relatively short period of time (corresponding to the frame length or the LCM), we next investigate how to allocate such energy budgets in the long run, so that to maximize the long-term (lifetime) system reward. We start by analyzing energy budget allocation for battery-powered embedded systems with a given battery charge in Section 5.3.1. Our analysis is then extended to a more complex scenario, in which the system alternates between recharging periods (where the system replenishes the battery at the same time as executing real-time applications) and discharging periods (where the system relies entirely on the stored energy in the battery) in Section 5.3.2. In particular, we propose energy budget allocation policies for maximizing the long-run system reward of embedded systems that use rechargeable energy.

5.3.1 Battery-Powered Systems

Let E_{max} denote the battery energy of a given system. While the previously proposed algorithms assume a given energy budget for a short period of time (such as in the milliseconds to seconds range, corresponding to the length of a frame), the battery capacity E_{max} determines the system lifetime, typically in the hours range. The problem is then how to allocate the short-term energy budgets E so that the long-term (lifetime) system reward is maximized.

Note that reward function are typically concave, while power functions are convex. This means that the reward of a task as a function of energy (assuming a fixed execution time) is concave. Extending to the entire system, the short-term system reward as a function of the short-term energy budget E (assuming a fixed frame length D) is a concave function. Thus, we observe that, given a long-term energy budget E_{max}, an energy partition into equal short-term budgets E will maximize the long-term system reward. The intuition behind this observation is based on the same argument used in [69], in which it is shown that, due to the concavity of reward functions and convexity of power functions, all task instances of a periodic task run at the same speed and for the same amount of time (thus, same energy) in the optimal (maximum reward) solution. If a frame is viewed as a periodic task (albeit

a more complex task), it follows that an equal energy partition between frames maximizes the system reward.

Thus, given a long-term energy budget E_{max} and a desired system lifetime t_{max}, each frame is allocated an energy budget $E = \frac{E_{max}}{t_{max}}$. Any of the previous algorithms, for both continuous and discrete reward models, can then be used to determine the schedule within a frame. An exception occurs when $E < E_{low}$, where E_{low} is the lowest energy budget that ensures a minimum reward solution. In this case, the frame energy budget is $E = E_{low}$ and the system lifetime is $D\frac{E_{max}}{E_{low}} < t_{max}$.

If there is no system lifetime requirement t_{max}, by the same concavity argument, the long-term reward is maximized when each frame is allocated an energy budget $E = E_{low}$, which results in a system lifetime of $D\frac{E_{max}}{E_{low}}$ time units.

Finally, note that the short-term reward maximization algorithms assume WCET for tasks. As average-case execution times are less than the worst-case, extra energy exists in the system that can be reclaimed to further improve the system reward. The issue of energy reclamation is discussed in detail in the next section, for the more general case of systems with rechargeable batteries.

5.3.2 Rechargeable Systems

While it would appear as though the device lifetime is ultimately dependent on battery storage capacity, some devices may scavenge the existing energy in the environment. An example of such a device is the NASA/JPL Mars rover, which relies on both a non-rechargeable battery source and a solar panel [52]. In this work, we assume that the battery is also rechargeable. During rechargeable periods (e.g., daytime for devices with solar panels) real-time tasks are executed at the same time the battery is recharging, while when the system cannot recharge, it relies entirely on the battery energy acquired during the recharging period. Since the periods in which recharging is possible may be limited, energy must be used efficiently.

5.3.2.1 Rechargeability Background A short introduction to rechargeable energy harvesting and storage (illustrated by a solar panel and respectively a rechargeable battery)

is given next.

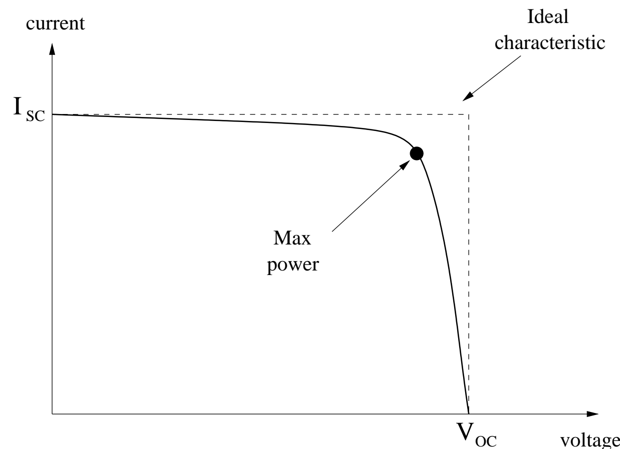
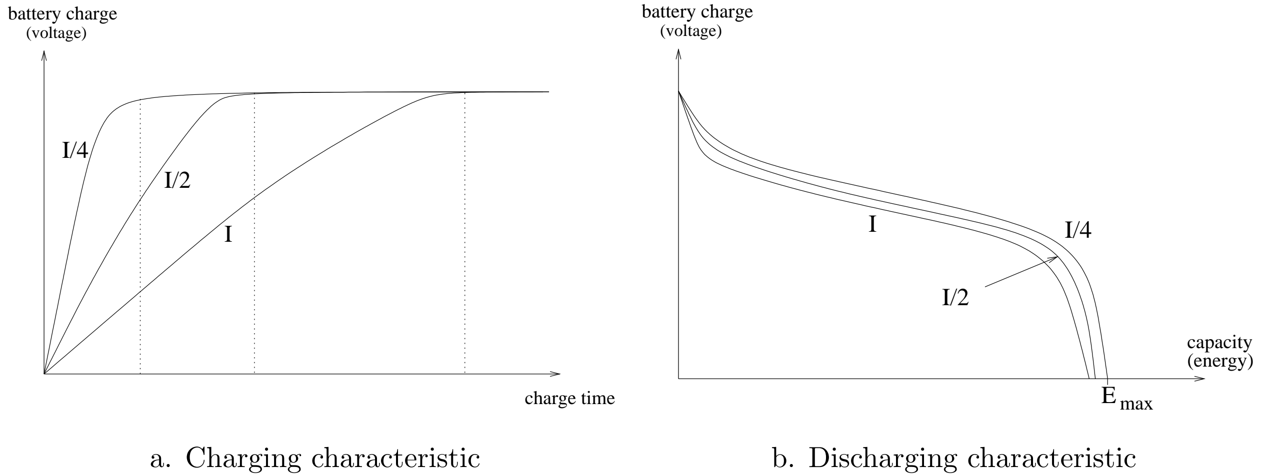


Figure 5.10: Current-voltage characteristic of a typical solar cell

A solar cell (also known as PV cell) converts light into electricity through the photovoltaic effect [85]. The current-voltage characteristics of a typical solar cell is shown in Figure 5.10. At short circuit the current is maximum (I_{sc}) but the power generated (the voltage multiplied with the current) is zero. Similarly, at open circuit the voltage is maximized (V_{oc}) but the current (and thus power) is zero. The optimal operating point (i.e., maximum power) is shown on the curve as P_{max}. Note that a solar cell cannot store energy by itself. The device attached to the cell will draw as much power as it needs; the remaining power (up to P_{max}) is simply wasted if not used.

A solar panel is obtained by connecting cells in series or parallel into PV arrays to obtain any desired voltage/current characteristic. Connecting two cells in series doubles the resulting V_{oc}; parallel connection doubles the resulting I_{sc}. For each cell I_{sc} depends on the intensity of light, while V_{oc} depends on other parameters (such as temperature).

A rechargeable battery has a nominal capacity (expressed in Amps-hour) corresponding to a maximum energy (expressed in Joules or Watts-hour). The charging characteristic for a typical lithium-ion rechargeable battery is shown in Figure 5.11a. The charging time depends on the charge current I, but also on other parameters (like temperature). Not all the power used to recharge the battery can be stored (for example, 1 W of charge for 1 hour results



a. Charging characteristic

b. Discharging characteristic

Figure 5.11: Charging and discharging characteristics of rechargeable batteries

in less than 1 Watt-hour stored energy). The discharging characteristic as a function of the discharge current I is shown in Figure 5.11b. The nominal capacity is computed for a given constant discharging current and temperature. A variable discharging current results in a reduced effective capacity.

5.3.2.2 Models and Problem Definition The recharging model and problem definition are presented next. The task model corresponds to multiple versions for frame-based and periodic tasks used in the previous sections. The problem solved is to determine how much energy to allocate to each frame so that to maximize the long-run system reward. *MV-Pack* is then used for maximizing the short-term system reward within a frame.

The system we target consists of three components: a processing unit, an energy harvester (such as a solar panel) and a rechargeable battery. The processing unit includes all the components needed for processing real-time tasks, such as a DVS processor, memory and network, and we assume that the task energy values $e_{i,j}^k$ refer to the consumption in the entire system comprised by the processing unit. The harvested power can be either used by the processing unit or stored for future use by the third component (rechargeable battery).

The shape of the solar power that can be generated on a satellite orbiting the Earth is shown in Figure 5.12a. The power is either constant (about 1350W/m^2) or zero if sunlight

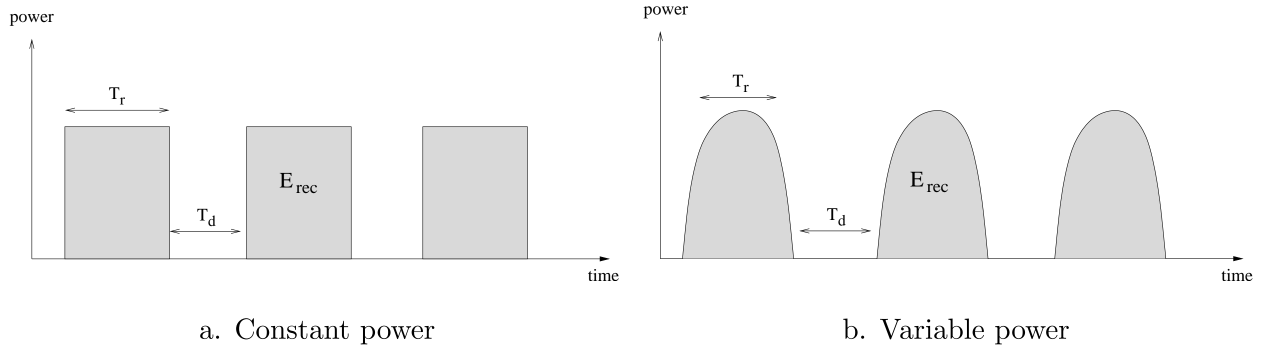


Figure 5.12: Solar power patterns

is obstructed [85]. We will refer to the time when there is solar power as T_r (recharging time). T_d (discharging time) denotes the time when the system has to rely entirely on the battery (i.e., no solar power). The amount of energy generated during T_r (the area below the power curve) is denoted by E_{rec}. Note that E_{rec} is only a fraction of the solar power, as the efficiency of a solar cell is typically 10% to 20%. The solar power on Earth's surface varies with time due to atmosphere and clouds, as shown in Figure 5.12b. For this scenario, E_{rec} will denote the worst-case amount of energy that can be generated during T_r.

We denote the maximum energy that can be stored in the rechargeable battery by E_{max}. As well, we denote by E_{min} the minimum energy required in the battery at any time. There is a loss of energy when recharging and discharging the battery. We use a parameter, $\alpha \in [0, 1]$, to denote the worst-case recharging loss. For example, for $\alpha = 0.9$ and $E_{max} = 9Wh$, $10Wh$ may be needed to fully charge the battery. A second parameter, $\beta \in [0, 1]$ denotes the worst-case discharging loss. Thus, if $\beta = 0.9$ and $E_{max} = 10Wh$, the actual energy is just $9Wh$ under a worst-case discharging scenario.

Our goal is to determine for each frame how much energy to allocate so that the system is stable (i.e., the battery energy can never be less than E_{min}), provided that E_{rec}, α and β, as well as task worst-case execution times and energy requirements are not underestimated. In addition, the energy is allocated taking into consideration task rewards, so that to maximize the long-term system reward (the sum of values for all versions selected for execution in all discharging and recharging frames). Using the multiple version task model of Section 5.2.2 and the *MV-Pack* algorithm, for a stable system, we also determine what task versions v_i to

select and at what speed levels s_i to run them. Note that $T_r >> D$ and $T_d >> D$, where D is the frame length. Thus, *MV-Pack* maximizes the short-term system value, given the energy budget allocated by the long-term energy allocation policy.

5.3.2.3 Static Energy Allocation We present necessary and sufficient conditions for the stability of the system. Based on these conditions we show how to distribute the available energy among frames, assuming a worst-case scenario.

The static analysis starts by running the *MV-Pack* algorithm, assuming infinite available energy. After each successful version increase (i.e., reward/value increase), the intermediate solution is saved (i.e., the speed and version for each task, as well as the total energy consumption and reward are stored). There can be at most NV successful version increases, thus the space and time complexity become $O(N^2V)$. In practice, running times are still under a millisecond even for 100 tasks (total running time in a Unix system with a 850MHz Pentium III CPU and 256MB of RAM).

The ith intermediate solution schedule, energy and reward are denoted by ϕ^i_S, ϕ^i_E and ϕ^i_R respectively. If a solution has a smaller reward and a higher energy than some other solution, it is eliminated from the saved solutions. This case can happen for artificial scenarios, although we did not encounter it during simulations. The saved intermediate solutions are ordered by their rewards/energy in increasing order. Note that even with infinite energy it may not be possible to run all the task at their highest version due to the real-time constraints. If the frame deadline is D, the number of frames to be executed during the recharging period T_r is $N_r = \frac{T_r}{D}$. Similarly, N_d denotes the number of frames to be executed during the discharging period, $N_d = \frac{T_d}{D}$.

Since it is expected that the frame reward increases less than linearly with the frame available energy, an equal energy partition is expected to maximize the total reward of the frames. Thus, we choose to distribute the energy equally among frames (recharging frame allocation may be different from discharging frame allocation due to recharging/discharging characteristics and battery capacity limitation).

We next identify necessary and sufficient conditions for a system to be stable (i.e., the battery energy is at all times above E_{min}). First, the generated energy during the recharging

period T_r must be enough to run all the frames with their minimum energy requirement ϕ_E^1. During recharging, the processing unit will use at least $N_r\phi_E^1$ energy. Due to the discharging loss β, at least $\frac{N_d\phi_E^1}{\beta}$ has to be stored for use during the discharging period T_d. Considering also the recharging loss α, the first condition for system stability is:

$$E_{rec} \geq N_r\phi_E^1 + \frac{N_d\phi_E^1}{\alpha\beta} \tag{5.33}$$

This condition is necessary but not sufficient, as it could be the case that not all of the recharging energy E_{rec} can be used (for example, due to battery capacity limitation). A second condition enforces that a fully charged battery holds enough energy to execute all discharging frames at their minimum energy consumption, even in worst case discharging conditions β:

$$E_{max} - E_{min} \geq \frac{N_d\phi_E^1}{\beta} \tag{5.34}$$

For a stable system, the actual schedules for the recharging and discharging frames are obtained as follows. We assume the system starts with a discharged battery (E_{min}) and the first recharging frame. The schedules for the recharging and discharging frames are the solutions ϕ_S^i and respectively ϕ_S^j that satisfy:

$$\text{maximize} \quad N_r\phi_R^i + N_d\phi_R^j \tag{5.35}$$

$$\text{subject to} \quad E_{max} - E_{min} \geq \frac{N_d\phi_E^j}{\beta} \tag{5.36}$$

$$E_{rec} \geq N_r\phi_E^i + \frac{N_d\phi_E^j}{\alpha\beta} \tag{5.37}$$

Determining the optimal values for i and j has complexity $O(NV)$, as there are at most NV stored solutions. A solution always exists for a stable system.

During discharging, the feasibility conditions 5.36 and 5.37 give the guarantee that the battery energy will never be less than E_{min} (i.e., the system is stable). During recharging, we assume the processing unit relies directly on the solar power, while the unused power is stored in the battery with a worst-case loss α.

5.3.2.4 Dynamic Energy Allocation Schemes The static policy (i.e., the solution to 5.35-5.37) is too conservative, as the system has to be stable even in worst-case conditions. Dynamic policies are proposed to handle cases when extra energy appears in the system. There are many ways to improve the system reward when worst-case scenarios do not happen. For example, whenever a task requires less energy than its worst-case, the remaining tasks inside its frame can benefit from the extra energy to improve their reward. However, this approach implies a considerable overhead as a new schedule needs to be constructed potentially every task completion. In terms of system reward the approach may also be inefficient, as it could be better to distribute the energy among frames.

Inspired from the work in [59], three dynamic policies are presented next. While the dynamic policies in [59] reclaim unused slack in order to improve the energy consumption, the resource reclaimed in our case is the energy itself, so that to improve the long-term system reward. We assume that the battery charge can be examined with reasonable accuracy. By inspecting the battery charge at regular intervals, dynamic schemes will observe the deviation from the worst case scenario and redistribute the available energy among frames so as to maximize the system value. Frame boundaries provide such regular intervals for checking the battery level. Thus, the extra energy is not used in the current frame and the rescheduling overhead occurs only at frame boundaries.

The first two schemes (Proportional and Speculative) redistribute the energy among all remaining frames until the first recharging frame (at which moment, because this is a stable system, the battery level is known to be at least E_{min}). Also, the system reward can benefit most from this approach since reward increases less than linearly with the energy. A third dynamic policy (Greedy) uses the static schedule for frames, but gives all the extra energy to the next frame. Rescheduling decisions are still made only at frame boundaries.

Proportional Energy Allocation In this scheme, upon the completion of each frame, the available energy is redistributed equally among all recharging frames and equally among all discharging frames. A worst case scenario is assumed for the remaining frames and thus the system is guaranteed to be stable. However, the extra energy can now be used to improve the system reward while still guaranteeing its stability. When recharging frame k completes,

aware of the current battery energy and the worst case remaining recharging energy, a new schedule ϕ_S^i is selected for the remaining $N_r - k$ recharging frames and a new schedule ϕ_S^j is selected for the N_d discharging frames so as to maximize $(N_r - k)\phi_R^i + N_d\phi_R^j$, while ensuring that the worst-case battery charge when the first discharging frame starts is enough to run all the discharging frames (i.e., is at least $E_{min} + \frac{N_d\phi_E^j}{\beta}$). Similarly, when discharging frame k completes, the available battery energy is equally distributed among the remaining $N_d - k$ frames so that the battery energy is at least E_{min} at the completion of the last discharging frame.

Speculative Energy Allocation The proportional scheme is too conservative as the worst-case scenario is assumed for all remaining frames. As has been shown in previous works [59], a better approach is to speculate about future energy consumption and schedule tasks accordingly, while ensuring that the system is stable even in worst-case conditions for all remaining frames. During discharging, the battery energy constantly decreases. At frame boundaries, the actual decrease in battery energy can be compared to the known worst-case. The ratio of actual consumption to worst-case consumption can be used to estimate consumption for future discharging frames. The ratio will be always less than 1, as the actual discharge loss is less than β and task energy consumptions will be less than their worst-case. The ratio for the next frame is then predicted as the average of such ratios for all frames in a history window. During recharging, a similar ratio is computed at frame boundaries for estimating the energy accumulating in the battery.

Greedy Energy Allocation This scheme assigns all the available extra energy to the next frame with the constraint that enough energy is left to run the remaining frames according to the static schedule. Thus, the extra energy in the system is immediately used, unlike in the previous schemes.

The overhead of all dynamic schemes is $O(NV)$ at each frame completion. Simulation results presented in the next section quantitatively evaluate both the static and the dynamic policies.

5.3.2.5 Experimental Results We simulated the Intel XScale architecture, with 5 speed levels (Table 5.1) for task sets with up to $N = 100$ tasks and $V = 4$ versions for each task. The execution times, energy values and task rewards were generated as described in Section 5.2.2.2. The static analysis was then performed by running the *MV-Pack* algorithm to generate the intermediate solutions ϕ_S^i.

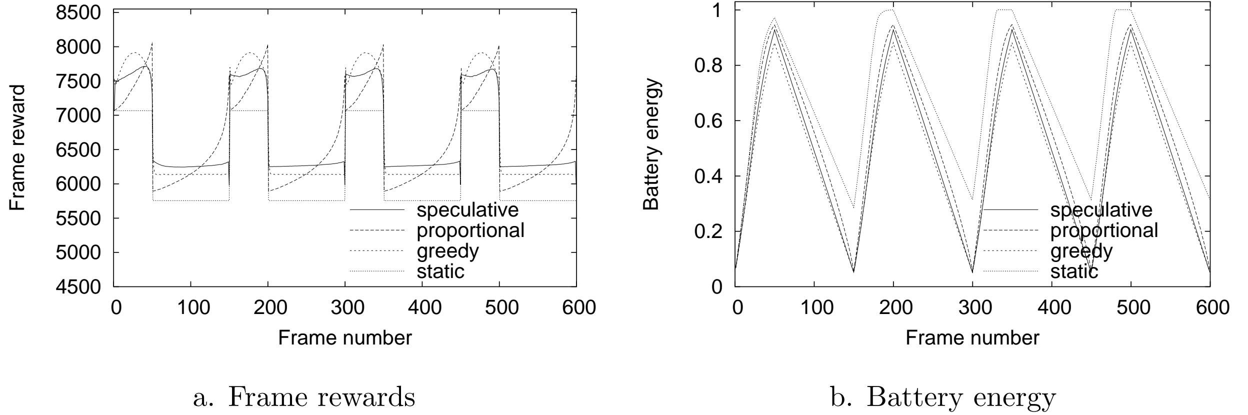
The values for E_{min}, E_{max} and E_{rec} were then generated as described next. E_{min} is 5% of the battery capacity E_{max}, which was generated so that the available energy during discharging (i.e., $E_{max} - E_{min}$) is at least $\frac{N_d \phi_E^1}{\beta}$ and less than $\frac{N_d \phi_E^s}{\beta}$, where s is the highest reward/energy intermediate solution (s is not always NV as deadlines can be missed). Thus, Equation 5.36 can be satisfied and not enough energy can be stored in the battery to run all tasks in all discharging frames at their highest version.

Similarly, E_{rec} was generated to be at least $(\frac{N_d}{\alpha\beta} + N_r)\phi_E^1$ and less than $(\frac{N_d}{\alpha\beta} + N_r)\phi_E^s$. Equation 5.37 can be satisfied, while the recharging energy cannot support all tasks at their highest version. We thus ensure that there is a solution, but not enough energy to run all the most valued task versions. We also ensured that the processing unit power during recharging periods is less than the worst-case solar power $\frac{E_{rec}}{T_r}$. Note that when the system has a large amount of energy (e.g. large N_r or large E_{max} and E_{rec}), the problem is unrealistic and uninteresting. The solution is also trivial, namely runs all tasks at the highest energy/reward solution s.

The static schedule was created as the solution to Equations 5.35-5.37. The system was then simulated, starting with the first recharging frame and a discharged (E_{min}) battery. The dynamic behavior is simulated as follows: with a probability of 50% tasks required their worst-case time and energy and with 50% probability their actual running time (and thus energy requirement) was between 50% to 100% of the worst-case. Thus, on average frames require 87.5% of their worst-case time and energy. We considered a worst-case α and β of 0.9. The actual α and β values were generated for each recharging/discharging frame in the range $[0.9, 1]$.

Note that the worst-case generated energy is $\frac{E_{rec}}{N_r}$ for each recharging frame, corresponding to the pattern in Figure 5.12a. To simulate a deviation from the worst-case, we added an extra energy of up to 20% for each recharging frame (i.e., the generated energy was in

the range $[\frac{E_{rec}}{N_r}, 1.2\frac{E_{rec}}{N_r}]$ for each recharging frame). To simulate the pattern in Figure 5.12b, we simulated the deviation from the worst-case as a sinusoidal function with a maximum of 20% in the middle of the recharging period.



a. Frame rewards

b. Battery energy

Figure 5.13: Static versus dynamic energy allocation policies

A comparison between the static and dynamic schemes for the recharging pattern in Figure 5.12b is presented in Figure 5.13. Results for the pattern in Figure 5.12a are very similar and are not shown. The frame reward and battery energy at the completion of each frame are indicated for all schemes. Each point in the graph is the average of 1000 experiments ($N = 50$ tasks, $N_r = 50$, $N_d = 100$). The overhead of redistributing the energy was typically in the range of microseconds to dozens of microseconds for each frame.

As seen in Figure 5.13b, the static scheme does not react to changes in the available energy and part of the recharging energy is wasted: the battery becomes fully charged before the recharging period ends (e.g., around frame 330). The dynamic schemes generally take advantage of all the recharging energy. In terms of frame rewards, all dynamic schemes outperform the static. Among the dynamic schemes the worst performance is that of the proportional, which is too conservative in assuming a worst-case scenario for the remaining frames. As a consequence, the available energy is too slowly redistributed, resulting in the pattern shown in Figure 5.13a, with frame rewards slowly increasing and extra energy accumulating towards the end of recharging and discharging periods.

The greedy scheme uses the extra energy immediately, with frame rewards following (on average) the sinusoidal shape of the extra energy during the recharging period. The speculative scheme only ensures that the remaining frames are feasible (i.e., minimum reward) and also speculates that tasks will not take their worst-case time and energy. For this reason it can be more aggressive and, on average, allocates more energy than the greedy policy to the discharging frames, and less energy to the recharging frames. Reducing the energy gap between the discharging and the recharging frames generally results in an improved total system value. The speculative scheme thus outperforms the greedy scheme in 83% of the experiments.

5.4 CHAPTER SUMMARY

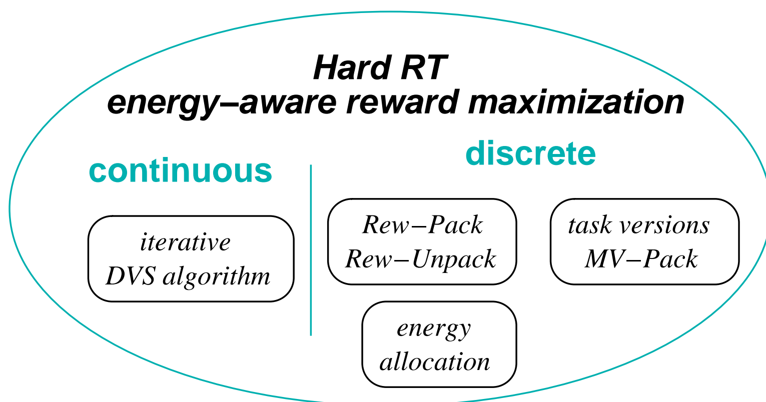


Figure 5.14: Chapter overview: reward maximization for hard real-time systems with energy constraints

The chapter summary is shown in Figure 5.14. The focus is on reward maximization schemes for hard real-time systems with energy constraints, corresponding to the first part of the research overview in Figure 3.2. We are interested in overloaded systems running adaptive real-time applications, and the problem is to identify the allocation of time to tasks (so as to respect the hard real-time constraints) and the DVS parameters for each task (i.e., the task frequency and voltage so that not to exceed a fixed energy budget).

We investigated these problems for two popular hard real-time application models: frame-based and periodic tasks. The problem formulation was shown to be equivalent under both models. For application adaptivity we use both continuous and discrete reward models. In the continuous case we use the known IC application model, while for the discrete model we propose to use task versions. The power models considered are both continuous and discrete. For continuous models we use the known formula expressing the processor power as a function of frequency and voltage, while for discrete power models we used published data for processors that support DVS.

We first proposed an iterative algorithm for the continuous case of power and reward models, as an extension to a previously proposed algorithm [5] that only considered two constraints at the same time. The algorithm adapts the solution to satisfy the system constraints by three mechanisms, referred to as *transfers*. The resulting system reward was found to be within 1% of the optimal (on average) in our experiments, with low running times.

We then directed our investigation to the more realistic case of discrete power and reward models. Two algorithms are proposed for single-version tasks, *REW-Pack* and *REW-Unpack*. The *REW-Pack* algorithm is based on the idea of task *packing*, which means sacrificing energy to improve the timing constraint. *REW-Unpack* performs the search for a solution in the opposite direction, sacrificing time to improve the energy constraint. Metrics that consider all three constraints (time, energy and rewards) are used throughout the algorithms. We show that the timing and energy constraints are equally important, and thus *REW-Pack* and *REW-Unpack* have similar performance. The two algorithms have running times in the microsecond range, with average errors within 3% of the optimal in our experiments.

As an extension to the discrete all-or-nothing reward model we propose the use of task versions to provide finer granularity. In this model each task may have several versions, with different computation requirements and rewards. The model is realistic and already applies to certain application areas that support different algorithms for each version. In addition, task versions already exist in a variety of forms, such as different parameters (for example or MPEG quality levels [89]), or different task invocation periods. The proposed *MV-Pack* algorithm is an extension of *REW-Pack* to handle such multiple versions. The

algorithm is shown to have similar performance and running times. In addition, it supports a combination of mandatory and optional tasks.

Finally, as the above algorithms rely on a given fixed energy budget, we propose a scheme to determine such budgets, so that to maximize the long-term system reward. In particular we investigate long-term energy allocation policies for battery-powered systems, as well as systems that use rechargeable energy. A static energy allocation policy assumes worst-case scenarios for task execution times and recharging parameters. Three dynamic energy reclamation policies are shown to further improve the long-term system reward, with overheads in the microseconds range at frame boundaries.

6.0 QOS-AWARE ENERGY MINIMIZATION FOR SOFT REAL-TIME SYSTEMS

In hard real-time systems, task arrival and execution times are known, leading to precise algorithms and power management schemes, as seen in the previous chapter. However, often there is no a-priori knowledge of the workload, corresponding to rate-based and aperiodic applications. Unpredictable workloads are characteristic to a variety of systems, ranging from cell phones and PDA devices to personal computers, servers and systems-on-a-chip [49, 80]. In addition, many such application domains are non-adaptive, meaning that there is no reward associated with individual requests. Instead, the system reward is defined in terms of QoS. The goal of such systems is typically not to maximize the reward, but rather to minimize the energy consumption while guaranteeing a required QoS.

Continuing the DVS work of Chapter 5 that targeted predictable real-time models, stochastic DVS schemes are first deployed for local power management of unpredictable workloads in Section 6.1. Stochastic DVS algorithms are more efficient than traditional prediction-based or utilization-based algorithms when prediction mechanisms fail to accurately describe the system load (as is the case for workloads with high variability).

We then direct our investigation to multi-processor systems, in particular server clusters. In these systems, a new problem is that of global power management (that is, turning on/off cluster machines) aware of QoS requirements. In addition, while the DVS power management algorithms proposed so far assumed a given load, a new challenge is how to distribute the load in a large multi-processor system. We first investigate load distribution mechanisms and power management policies in the context of servers and homogeneous server farms in Section 6.2. The work is then extended with QoS-aware local and global policies for the general case of heterogeneous systems in Section 6.3.

6.1 STOCHASTIC DVS FOR UNPREDICTABLE WORKLOADS

This section evaluates dynamic voltage scaling (DVS) policies for power management in systems with unpredictable workloads. A clear winner is identified, namely a stochastic policy that reduces the energy consumption one order of magnitude compared to no power management, 50% on average in synthetic workloads, and up to 40% for real-life traces compared to the second-best evaluated scheme.

6.1.1 Two Request-Driven Signal-Processing Applications

The examples that prompted our investigation comes from our industrial research partners, dealing with satellite-based signal processing. Signal data collected through an external sensor is disseminated to several processing units for further analysis. A signal processing application is responsible for timely analysis of the signal data (also referred to as requests, or events). We are investigating two such applications, known as Subband Tuner (SBT) and Complex Ambiguity Function (CAF), each provided with several realistic traces.

SBT is an application that searches digital signal data that is related to the frequency and time domains for certain patterns. It uses filters for finding contiguous chunks of data that have a specific characteristic for a certain interval of time. After finding such patterns, there is some processing that occurs. There are two possible paths to be followed, a short and a long path, depending on the type of the event. From our own measurements, in about 21% of the events, there are not enough details to quickly extract the correct data and extra processing is required. For the other 79% of the events, the data is sufficient for quick processing.

CAF is an application that collects data in low orbiting satellites (LEOs) and correlates it with data collected from geo-stationary satellites (GEOs), for object recognition and location. CAF processing is done in the LEO through calculations of the difference between arrival time (dT) and frequency (dF) signals from the object of interest. The object may be on Earth's surface or may be flying. The CAF application can determine an object's location with an accuracy from 4 to 7 significant digits (corresponding to 1K to 16K data point

correlation, respectively).

For these applications, the major goal of the power-management policy is to keep up with the rate of request arrivals, while minimizing the energy consumption. In addition, the processing of requests is expected not to exceed a maximum response time (soft deadline).

Tables 6.1 and 6.2 present statistical information about request execution and inter-arrival times for two SBT traces and one CAF trace. As it turns out, the request execution times, as well as request inter-arrivals, are rather unpredictable, with large deviations from averages.

Table 6.1: Request execution times (in millions of cycles)

	SBT		**CAF**		
	Type 1	Type 2	Type 1	Type 2	Type 3
Min	2.9	2.0	8.2	4.1	1.3
Max	82.6	753.6	5045	210.2	32.9
Avg	9.7	123.2	820.2	45.0	5.8
Stdev	7.2	153.8	1251	78.5	6.2
%	79%	21%	5.4%	2.9%	91.7%

Table 6.2: Request inter-arrival times (in seconds)

	SBT		**CAF**
	81 min	1030 sec	1800 sec
Min	0.13	0.1	0
Max	6.7	11	5
Avg	0.37	0.44	0.7
Stdev	0.62	0.77	1.74
events	13045	2307	2564

Note as well that each application has several types of requests/events. The SBT application can determine solely from processing the header of a request if its computation is expected to be relatively short, or if there is a possibility that it may take much longer (thus, two types). For the CAF application there are three types: the first time an event occurs, the second time and all times after that (typically resulting in long, short to medium, and relatively short events, respectively). Request types are the sort of semantic information that helps improving the predictions about the workload.

6.1.2 System Model

Requests are processed on a DVS processor, with M discrete operating frequencies. The average power consumption of the system is known at each operating frequency/voltage. We denote the average power consumption at frequency f_i by P_i. Using IBM's Mambo cycle-accurate and power-accurate simulator for the PowerPC platform [78], we observed very little variation in power within applications at a constant frequency/voltage, for both SBT and CAF. Thus, describing energy consumption using average power values for each frequency/voltage results in a precise estimation. The exact values used (for the PPC405LP processor) are shown in Table 6.3. Both time and energy overheads are considered for speed changes. The time overhead in our experiments is 1ms, and the power during speed changes is P_M (that is, the power at maximum frequency).

Table 6.3: PPC405LP power consumption of SBT and CAF applications

Speed (MHz)	33	100	266	333
Voltage (V)	1.0	1.0	1.8	1.9
Power (mW)	19	72	600	750

Requests are generated externally and buffered in the system memory for further processing. Once arrived, a request must finish processing by its deadline D (different requests may have different deadlines). Deadlines are soft, meaning that occasional deadline misses, while undesired, do not result in system failure. To reduce the number of deadlines missed, a good DVS policy typically chooses the maximum speed whenever it is possible that a request may miss its deadline.

Requests are scheduled on the processor in a first-come first-served (FCFS) fashion, without preemption, as subsequent requests may depend on results from previous requests. The DVS algorithm is invoked in one of two situations: when a timeout expires, or when an event specifically requests a speed change. The timeout mechanism can specify exactly at which moment in time to change the speed. For example, in our stochastic scheme two speeds are used for task execution, a primary and a secondary speed. When a timeout (determined based on the number of cycles at the primary speed) expires, the task moves from its primary (base) speed to its secondary (backup) speed. In addition, two events may

trigger a speed change: the arrival of a new request, or the completion of the current request. The goal of the DVS algorithm is to select the minimum speed (i.e., minimum energy) that will not cause requests to miss their deadlines.

6.1.3 Prediction Schemes

Prediction-based DVS algorithms adjust the performance based on workload estimations, which, in turn, are based on the execution history. The CPU speed is adjusted based on predicted resource requirements. Thus, the success of such schemes depends on how accurately the future workload can be predicted.

6.1.3.1 Application-Oblivious Prediction (AO) The simplest form of prediction is one that only monitors CPU utilization, unaware of the applications running on the system. CPU utilization (monitored periodically and defined as CPU busy time over total time) is a great indicator of past resource usage. The CPU speed for the immediate future is then increased or reduced based on utilization trends. Without a complex prediction scheme, the underlying assumption is that the immediate future resembles the immediate past.

Our application-oblivious prediction scheme works as follows: the system utilization is continuously monitored and checked periodically, every p time units. If the system was fully utilized during the last monitored period, the speed is increased to the next higher discrete frequency. If the utilization u is less than 100%, the CPU speed s is updated as $s = \lceil s \cdot u \rceil_f$, where $\lceil x \rceil_f$ is the function that returns the smallest discrete frequency higher than x.

We experimented with many other variations on this scheme, such as using utilization thresholds to determine when to increase or reduce the CPU speed. An example of such a scheme for saving energy in web servers is mentioned in [10], where utilization was monitored every 20 milliseconds. Whenever the CPU utilization exceeds 95%, the CPU speed is increased to the next higher level. Similarly, when utilization falls below 80%, the speed is decreased one level. Another example of the utilization policy is Transmeta's firmware implementation (LongRun) [21]. CPU utilization is frequently monitored, resulting in performance speed-up/slow-down by one performance level.

The main problem with utilization-based policies is that they use a fixed monitoring period for the system lifetime. For workloads with large variations, this results in the system being either too aggressive or too slow to react. When the monitoring period is too small, the system may set the speed higher than necessary. If the utilization is monitored too infrequently, requests may have large response times. Whenever the monitoring period does not correspond to the workload, utilization-based schemes may not be efficient. Accordingly, a software DVS algorithm [32] was shown to achieve 11%-35% more performance reduction (and thus energy savings) over LongRun. Another disadvantage of utilization-based policies is that request deadlines are not considered. On the other hand, the main advantage is simplicity. The policy does not require anything more than a timeout mechanism and a way of monitoring the system utilization. Even better, some CPUs already provide this functionality.

We also note that utilization need not refer only to the CPU. Other resources (hardware or conceptual) can be monitored. For example, we experimented with monitoring the number of requests buffered for processing in memory. An increasing/decreasing number of waiting requests indicates that there may be a need for higher/lower speeds. Monitoring these other resources allows optimization of the system based on those resources. Another example is when memory banks can be turned off: we would like to limit the number of requests that can fit (most of the time) in a single memory bank, and so monitoring memory usage may be useful.

6.1.3.2 Application-Aware Prediction (AA) Rather than simply reacting to observed resource requirements, AA schemes attempt to predict future performance needs by monitoring request inter-arrivals and processing requirements. As with utilization-based schemes, the CPU speed is adjusted periodically, with a pre-specified period (or timeout) p. Every p time units, the number of requests (of each type) arriving in the next period is predicted based on recent such information. The average execution time of each request is similarly predicted. We also studied schemes that adjust the speed at irregular intervals corresponding to request completion times and to the arrival of a certain number of requests in the system, and noted that a periodic adjustment scheme results in more energy savings.

Let N_i denote the predicted number of requests of type i arriving in the next period, each of which has average execution time (at maximum speed) a_i. Further, let L_i denote the number of requests of type i unfinished from the previous period. $\sum_i (L_i + N_i)a_i$ represents the predicted amount of work for the next period p. The speed s is recomputed as $s = \lceil \frac{\sum_i (L_i+N_i)a_i}{p} f_M \rceil_f$.

L_i is known and N_i is predicted as the number of requests in the last period. a_i is predicted in one of three ways: the average for the whole trace (from offline profiling), the average of the last period, or a combination of the two (such as exponential decay).

The above formula guarantees that the system will keep up with the rate, as it attempts to complete all waiting requests in the next period. However, since these schemes ignore the deadlines, the penalty incurred in missing deadlines may be very large. As expected, considering semantic information (i.e., request types) generally improves the quality of the prediction.

The efficiency of prediction-based DVS schemes greatly depends on the prediction accuracy. For unpredictable workloads, the schemes may be too aggressive or too slow to react, depending on the period p. As with utilization-based schemes, no request deadlines are considered. Moreover, implementation of the prediction policy is required.

6.1.4 A Stochastic DVS Algorithm

While stochastic schemes still collect statistical information about the workload (a-priori or online), they differ from prediction-based schemes in that they do not attempt to predict request processing requirements and inter-arrival times. The data collected is the probability distribution of request CPU cycles. Requests are classified based on their number of cycles, with any desired granularity. For example, with a granularity S (expressed in cycles), class C_0 would contain all requests whose cycles are up to S, class C_1 contains all requests with cycles in the range $(S, 2S]$ and class C_i represents requests with cycles in the range $(iS, (i+1)S]$. Counting the number of requests belonging to each class results in a histogram with $B = \lceil \frac{WC}{S} \rceil$ bins, where WC denotes the worst-case number of cycles of a request. We store the histogram as an array H of size B, where $H[i]$ denotes the number of request in class (bin,

or position) i.

The probability distribution can be obtained from profiling or through on-line monitoring, as follows: every time a request finishes processing, its exact number of cycles e is known. To include the request in the histogram, its corresponding class count is updated as follows: $H[\lceil e/S \rceil] = H[\lceil e/S \rceil]+1$. Of course, separate histograms can be maintained for each request type.

The cumulative density of probability function *CDF* associated with a histogram is defined as $CDF[k] = \frac{\sum_{i=1}^{k} H[i]}{\sum_{i=1}^{B} H[i]}$ (i.e., the probability that a request requires less than kS cycles, or the probability that a request belongs to one of the first k bins). When a new request enters the system, the function $1 - CDF[k]$ represents the probability that bin k will be executed (i.e., the request will probably execute for at least kS cycles).

We start by showing how to use DVS to minimize the expected energy consumption of a *single* request with a known histogram and deadline. It was previously shown that an optimal DVS schedule would gradually increase the speed [54, 36, 95]. While the work in [54] is only intended for continuous speeds, an exact solution for specific power functions is proposed in [36] and [95]. Since we are interested in systems with more general power functions (e.g., those that include other components beside the CPU), we are proposing a simple, novel DVS scheme that selects just two speeds among M possible speeds for each request: a primary and a secondary speed.

Our scheme chooses a primary speed f_i and a secondary speed f_j so that to minimize the expected energy consumption. If WC is the worst-case number of cycles and D is the request deadline, and $WC/D \geq f_M$ (i.e., the worst-case number of cycles cannot be satisfied within the deadline even at maximum speed), the scheme selects $f_i = f_j = f_M$. If $WC/D \leq f_1$, then $f_i = f_j = f_1$. Otherwise, the expected energy for all combinations of primary and secondary speeds $f_i \leq WC/D$ and $f_j \geq WC/D$ is computed as follows.

The time spent at the primary and secondary speeds, t_i and t_j, is first determined by solving the linear system described by the equations $t_i f_i + t_j f_j = WC$ and $t_i + t_j = D$, as in [45]. The first $t_i f_i$ cycles are executed at the primary speed, and the remaining cycles (up to WC) are executed at the secondary speed. The expected energy consumption of bin k is $E_k = P_i \frac{S}{f_i}(1 - CDF[k])$, if $(k+1)S \leq t_i f_i$ or $E_k = P_j \frac{S}{f_j}(1 - CDF[k])$, if $(k+1)S > t_i f_i$.

The total expected energy consumption is $\sum_{k=1}^{B} E_k$.

Depending on the probability distribution of request processing requirements, the primary and secondary speeds can differ by more than one discrete level. Because we consider probabilities, our solution is different from [45] (and other subsequent work), which picks adjacent speed levels for f_i and f_j. This is precisely the intuition behind the stochastic approach: if most requests are short enough to finish execution at the primary speed, a larger gap between the primary and secondary speeds will result in more savings compared to the adjacent speeds.

The difference is larger for distributions where the worst-case is much higher than the average case, such as bimodal distributions. After computing all combinations of primary and secondary speeds (at most $M^2/4$, or four combinations for $M = 4$ discrete speeds), the one with the smallest expected energy consumption is selected as the final DVS schedule.

A straightforward implementation of the above DVS algorithm has complexity $O(BM^2)$. From our experience, $B = 100$ bins results in a good enough representation of the histogram. Combined with a small number of discrete speeds M, this leads to a very efficient algorithm. Note that the complexity can be improved to $O(B + M^2)$ if using $O(B)$ extra space. This low complexity can be accomplished by storing a cumulative *CDF*, defined as $CCDF[k] = \sum_{i=1}^{k} CDF[i]$, which requires $O(B)$ extra space. Using the *CCDF*, the summation in $\sum_{k=1}^{B} E_k$ can be transformed into a $O(1)$ computation. Furthermore, if the histogram is collected offline (i.e., no need to update the *CDF*), the complexity becomes just $O(M^2)$.

The description above considered only a *single* request. With *multiple* requests dynamically entering the system, we extend our algorithm as follows: aware of all waiting requests and their deadlines, the latest completion time of the first request is first computed, so that no request can miss its deadline, even in worst-case scenarios (i.e., assuming that all requests take their worst-case number of cycles). That is, with new requests arriving with their own deadlines, the DVS algorithm may have to consider an artificially-reduced deadline for the current request, to ensure that enough time remains for the queued requests (at maximum speed) to meet their own deadlines. The approach is greedy, as it assumes maximum speed for the queued requests in order to create slack for the current request. At the same time the approach is conservative, as it assumes worst-case execution times. Note that the first

assumption does not imply that all subsequent requests will run at maximum speed, as in practice average processing requirements will be less than their worst-case. Whenever a request finishes execution, the slack created can immediately be used by the next request. The DVS schedule is then computed as described in the single request system.

The complexity of computing the latest completion time of requests is $O(N)$, where N is the number of requests queued for execution. The artificially-reduced deadlines are computed assuming maximum speed and worst-case execution times, in reverse order of the queued requests, to ensure that all deadlines are met. Note that the computed latest completion time may be less than the request deadline, to accommodate for processing requirements of subsequent requests.

Speed change overheads are considered when recomputing the DVS schedule. Also, whenever a new request comes, a better estimation is obtained for the current executing request by considering only the remaining cycles when calculating the expected energy. For each request, the number of speed changes is at most M. Experimental results in the next section will show that the number of speed changes is much less in practice.

6.1.5 Experimental Results

The traces shown in Tables 6.1 and 6.2 are evaluated next on the PPC405LP power model in Table 6.3. The monitoring period for the prediction-based schemes (AO and AA) is $p = 1$. The stochastic scheme (S) does not need a monitoring period, as it makes no prediction about the workload. Instead, a soft deadline is used as the maximum allowed response time for requests. The embedded applications we are dealing with have the same deadline for all requests. Since the worst-case execution time (at maximum speed) is 2.3 seconds for SBT and 15 seconds for CAF, we show results for deadlines equal to approximately twice and four times the worst-case. The histogram for each request type was obtained offline. We used a fixed bin width of 10 million cycles, resulting in 76 bins for SBT and 505 bins for CAF. The corresponding space overhead for the histograms is 304 bytes for SBT (2 types, 76 bins, 2 bytes for each bin) and 3030 bytes for CAF (3 types, 505 bins, 2 bytes per bin).

In all the schemes (except no-power-management), the system immediately switches to

minimum speed when idle (i.e., $P_{idle} = P_1$). A time overhead of 1 millisecond is added for each speed change. We compute the energy overhead of a speed change as the system power at maximum speed times the time overhead. We note that both the time and energy overheads do not have a significant effect for our traces, due to infrequent speed changes (less than two speed changes per second for all schemes). The overhead of the policies themselves is in the microseconds range for each speed computation (at most two speed computations per request are necessary for the stochastic scheme).

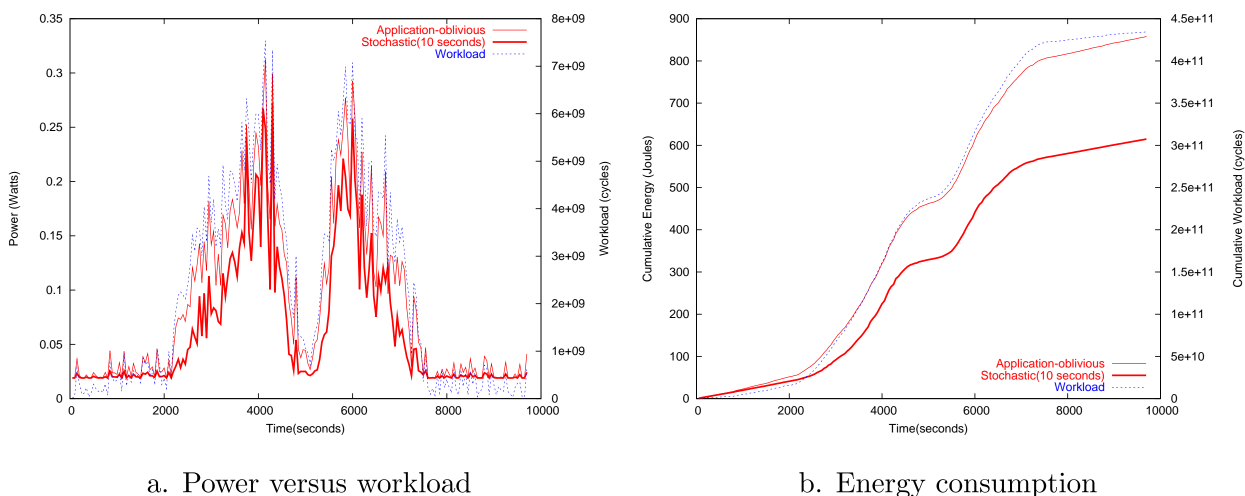
Evaluation results are shown in Table 6.4 for the prediction-based schemes (AA and AO) and the stochastic scheme (S) with two different deadlines (5 and 10 seconds for SBT, and 30 and 60 seconds for CAF). Two traces are considered for SBT (with lengths of 81 minutes and 1030 seconds respectively), and one trace for CAF (1800 seconds), as described in Tables 6.1 and 6.2. To vary the system load for the same trace, the original inter-arrival times between requests are multiplied with a *scale factor*, as in [26]. We considered the following values for the scale factor: 0.8, 1, 2, and 4, where 1 is the scale factor for the original trace. Reducing the scale factor below 0.8 results in a overloaded system that cannot keep up with the rate even if running at the maximum speed at all times.

The savings, shown in columns 2-5 of Table 6.4 for each scale factor, are normalized to the no-power-management (noPM) scheme. The stochastic approach results in the most energy savings, up to 28.5x compared to no-power-management and up to 40% less energy compared to the second-best DVS scheme (the application-oblivious prediction). The stochastic scheme also results in the fewest speed changes per second (SC/s) among the DVS policies (see column 6, Table 6.4). This is because many requests do not reach the point where they switch to the secondary speed. Also, when the system is overloaded or under-utilized, the primary and secondary speeds are identical (i.e., f_1 for a under-utilized system and f_M for an overloaded system).

In most experiments there were no deadline misses (average and maximum delays for the scale factor of 1 are shown in the last two columns of Table 6.4). We also experimented with tighter deadlines and noted that the maximum delay of the stochastic scheme closely matches the specified deadlines (unless the workloads is so high that the deadlines cannot be met).

Table 6.4: Evaluation of DVS policies for unpredictable workloads

policy	Savings (per scale factors) 1	2	4	0.8	SC/s	Avg delay	Max delay
81 min (SBT)							
noPM	1x	1x	1x	1x	0	0.43	5.22
AA	3.8x	7.5x	13.2x	3x	1.27	0.72	5.43
AO	4x	8.4x	16x	3.1x	1.77	0.86	5.45
S 5s	4.4x	10.5x	21.2x	3.4x	1.05	1.42	5.88
S 10s	4.6x	11.7x	26.7x	3.5x	1.08	2.78	9.63
1030 sec (SBT)							
noPM	1x	1x	1x	1x	0	0.36	2.51
AA	4.4x	8.5x	14.5	3.5x	1.1	0.67	3.00
AO	4.6x	9.7x	17.5x	3.7x	1.46	0.90	4.31
S 5s	5.4x	12.5x	23.1x	4.1x	0.95	1.4	4.5
S 10s	5.9x	14.8x	28.5x	4.3x	0.91	2.88	9.01
1800 sec (CAF)							
noPM	1x	1x	1x	1x	0	1.54	29.90
AA	4.3x	7.7x	12.9x	3.5x	0.6	1.58	29.89
AO	4.7x	8.5x	14.2x	3.8x	0.39	2.37	31.68
S 30s	4.9x	9.6x	18.2x	3.9x	0.11	3.06	34.41
S 60s	5.2x	11.6x	23.4x	4x	0.07	5.65	42.06



a. Power versus workload

b. Energy consumption

Figure 6.1: Power and energy consumption for the SBT 81 minutes trace

As seen in Figure 6.1a for the 81 minutes SBT trace, the system power closely matches the workload (cycles per second), with lower power consumption for the stochastic policy. The cumulative energy consumption for the same trace is shown in Figure 6.1b. Similar results hold for the other two traces.

Finally, we evaluate the DVS policies on a set of synthetic traces. The purpose of these experiments is to compare the AA, AO and S schemes for more request distributions (that is, in addition to those described in Tables 6.1 and 6.2). For the request cycles we artificially generate three typical distributions, namely, bimodal, normal (Gaussian) and uniform distributions, each with a minimum of 5 million cycles and a worst-case of 200 million cycles (20 bins). We use a uniform random number generator to generate the arrival rate for each second, which results in a unpredictable workload. The deadline (maximum response time) for the stochastic scheme is 5 seconds. For each experiment, the average system load is around 30% of the maximum load that the system can handle. Note that 30% is actually a high load, and parts of the trace result in a large request queue (that is, many requests are waiting for execution). For example, a long request takes 6 seconds at 33MHz, causing subsequent requests to miss their deadlines. Figure 6.2 shows the average energy consumption of the DVS policies (1000 experiments were averaged for each distribution). The stochastic scheme achieves up to 20x savings compared to no-power-management and, on average, 50% more savings compared to the second-best scheme (AO).

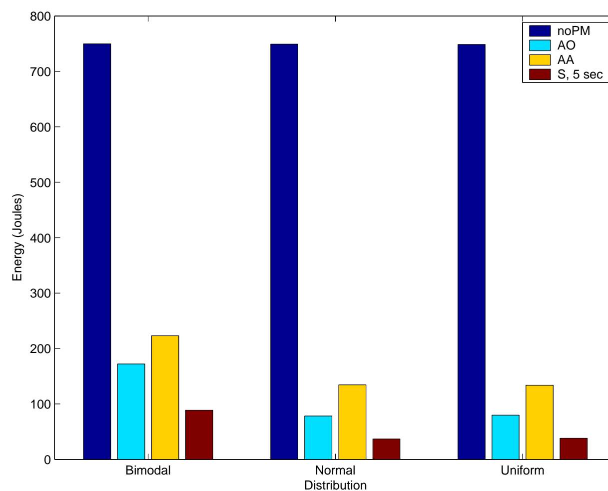


Figure 6.2: Comparison of the DVS policies using synthetic traces

6.2 HOMOGENEOUS REAL-TIME CLUSTERS

While so far we presented work dealing with single-processor systems, the remaining of this chapter is dedicated to the power management of multi-processor systems, in particular server clusters. Two additional challenges emerge in such systems: (a) load distribution in a multi-processor environment and (b) global power management (i.e., turning on/off cluster machines as required by the load). This section proposes a global load-aware power management scheme for homogeneous clusters. The work is then extended with global and local power management policies for the general case of heterogeneous systems in Section 6.3.

6.2.1 Cluster Model

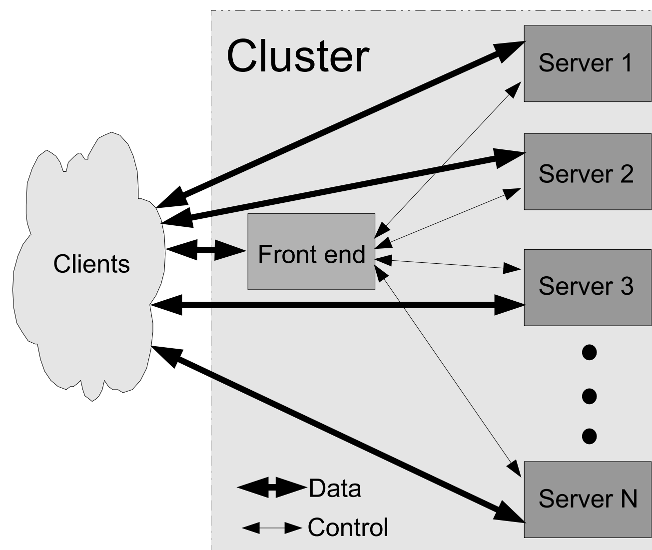


Figure 6.3: Cluster architecture

The cluster model is shown in Figure 6.3. The cluster consists of a front-end and N identical servers (nodes), each equipped with a DVS processor. At any given time, each node is in one of three states: active, idle, or inactive (off). When a node is active, its processor is running at some frequency f_i, and the power consumption is P_i. The idle power consumption is P_{idle}, and the power consumption when inactive is P_{off}.

The front-end is responsible for collecting requests from clients and for distributing the requests to the active servers. The applications considered are SBT and CAF, introduced in the Section 6.1.1. The system load is the amount of work (in cycles) that the front-end receives in one second. The front-end is also capable of turning on/off servers, according to some policy. The front-end attempts to distribute the incoming requests among the active nodes in a balanced fashion.

Each active node carries out DVS independently, running at the lowest frequency that keeps up with the request arrival rate. Upon servicing the requests allotted by the front-end, results are returned directly to the clients. Feedback is also sent to the front-end, such as speed changes.

6.2.2 Load-Aware On/Off DVS (LAOVS)

The power management policy is load-aware and combines local power management (DVS) with global schemes (on/off). Example of local DVS algorithms were given in Section 6.1. In this section we focus on the front-end algorithms, that is: how to estimate the system load, when to turn nodes on and off, and how to distribute requests to active nodes.

6.2.2.1 Load Estimation We determine the number of active nodes based directly on the system load, which can be obtained by getting feedback from each node. This is in contrast with the policies in [26], which turn on/off nodes based on the frequency of active nodes. The assumption in [26] is that the active nodes are not idle most of the time (continuous frequencies) and that they are all running at the same frequency (perfect load-balancing). However, the frequency of a node at a given time does not necessarily correlate well with its actual load (unless the load is well balanced and the frequency is continuous), and can result in a poor estimation of the average frequency needed.

In our policy, we determine the number of servers needed based directly on the load in the recent past (rather than the node frequencies). This results in a more accurate estimation and eliminates the strong dependency on perfect load balancing. The number of nodes needed for each load is computed offline using simple approximation algorithms, and

is stored in a discretized table. The number of nodes is computed without the assumption of continuous frequencies.

At runtime, the front-end records the history of recent speed changes of each node, which means that each node needs to send the history of its speed changes as feedback to the front-end. For each node, the front-end computes the average load over the past *look_back* seconds, where *look_back* is a system parameter. Note that the average load is different from the average frequency because idleness is regarded as frequency zero, while the CPU is operating at some frequency when idle. The summation of the average loads of all nodes is the estimation of the system load. The number of servers needed is then decided by a simple table lookup.

The estimation of the system load and deciding the number of active nodes based on the system load are the highlights of our global on/off policy. Since the global policy uses the system load and the local server policies rely on DVS algorithms, we call our integrated policy Load-Aware On-off with independent Voltage Scaling (LAOVS).

6.2.2.2 Threshold Schemes After the front-end obtains the estimation of the system load, it determines the number of active nodes needed, n_o by table lookup. Let the current number of active nodes be denoted by n_c. If $n_o = n_c$, there is no need to make any nodes active/inactive; if $n_o > n_c$ and $n_c < N$, make an inactive node active; if $n_o < n_c$ and $n_c > 1$, make an active node inactive. To be conservative, we do not make more than one node active or inactive at a time.

In real-life workloads, there may be some short-lived temporary workload changes. This may force the front-end into making a node active or inactive if we do that once the front-end detects a workload change. To prevent this from happening, we design a threshold scheme. We define two variables: *shutdown_threshold* and *restart_threshold*. The variable *shutdown_threshold* is the time the front-end will wait before it is sure to make a node inactive. Similarly, *restart_threshold* is the time the front-end will wait before it is sure to make a node active. During the waiting time, the front-end will continue tracking the system load. At the end of the waiting time, an active node will be made inactive only if the decision to turn off a node is true every time it was checked during the waiting time.

Similarly, an inactive node will be made active only if the decision to turn on a node was true every time it was checked during the waiting time.

6.2.2.3 Workload Distribution Since neither the request arrival rate nor the computational requirement for each request is known a-priori, the front-end approximates the load-balancing by sending the next request to the active node with the lowest average frequency over the past *look_back* second(s) [12].

Let AN be the number of nodes in the active state. To simplify the tracking of the status of the nodes, our policy is to keep nodes $1, 2, \ldots, AN$ active and node $AN + 1, \ldots, N$ inactive. The value of AN is at least 1 which means node 1 is always active. Under this policy, when we want to make a node active, node $AN + 1$ is the target; when we want to make a node inactive, node AN is the victim. This implementation was preferred because of its great simplicity, with complexity of $O(1)$.

Table 6.5: IBM PowerPC 750 frequencies and system total power (measured)

f(MHz)	idle	4.125	8.25	16.5	33	99	115.5	132
P(mW)	1150.0	1150.0	1369.0	1811.0	2661.0	4763.0	5269.0	6533.0

6.2.2.4 Experiments on a Testbed As a proof of concept, to build a real system in a hardware platform, we implemented our scheme on a testbed consisting of IBM PowerPC 750 nodes. The scheme was implemented in a joint effort with Ruibin Xu and Dakai Zhu; additional theoretical results and thorough simulation-based evaluation are presented in [93].

Power characteristics of the PPC750 system are presented in Table 6.5. This platform, provided by our industrial partners with round-robin (RR) as the request distribution policy at the front-end, only had 2 nodes available at the time of testing. In these experiments our goal was not to do extensive evaluation in comparison to other schemes, but to implement a proof-of-concept system, to take actual measurements, and to evaluate power management policies to a certain extent. Our collected data can be seen through our web interface [62].

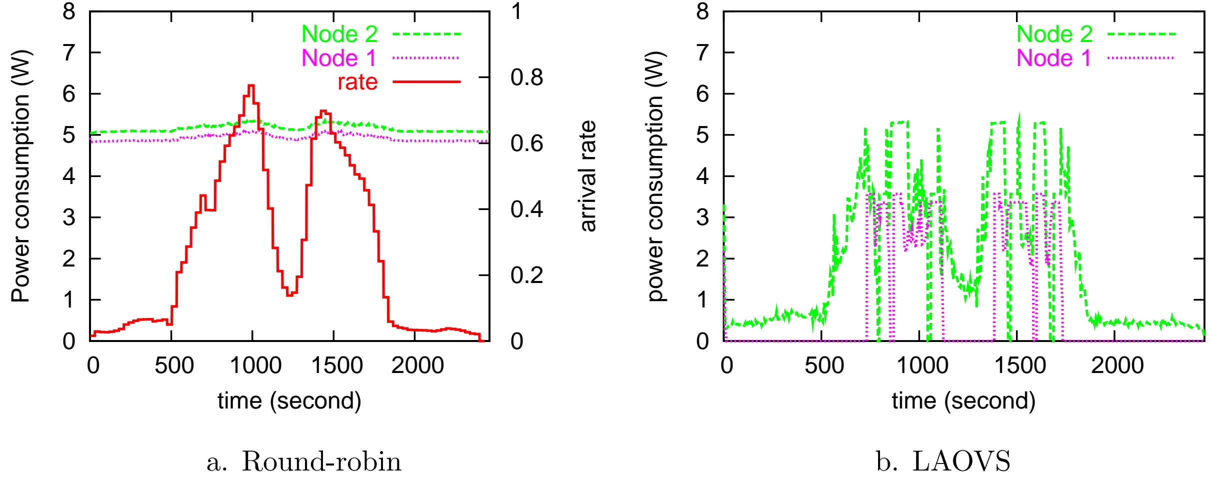
The voltage/frequency scaling of PowerPC 750 processors is done through external voltage/frequency regulation. The voltage scaling takes around $2ms$ for each $0.1V$ adjustment

and the frequency scaling takes negligible amount of time. Linux is used in the nodes and the power management (DVS and on/off) is supported through APIs. To power on/off a node, a specific message is sent to a host machine that has been wired to all nodes. Powering off a node takes effect immediately while powering on a node (booting from ROM including uncompressing the kernel) takes 33 seconds. The power consumption of each part in a node (e.g., processor, memory and I/O) was measured and collected using a data acquisition system. The variable *look_back* is set to 4, which means the front-end computes the average load over the last 4 seconds. Both *shutdown_threshold* and *restart_threshold* are 4 seconds. The front-end checks the system load every 1 second, deciding whether to turn on or off a node.

In our experiments, we used an additional node to be the front-end, which emulates the sensors that generate events; we report results of the 81-minute trace of SBT. The front-end sends events to nodes every second. On each node, a self-adaptive DVS scheme is employed, which uses the utilization information during the last interval (i.e., one second) to determine the frequency/voltage level for the next interval. For our LAOVS policy, whenever a node changes its voltage/frequency or becomes idle, a feedback message is sent to the front-end.

Figure 6.4a shows the results when the RR policy is used at the front-end and both nodes are always kept on. The solid line in Figure 6.4a is the normalized arrival rate (Y-axis on the right side of the plot) of events used in our experiments. Notice that the arrival rate is a rough indicator of system load. The arrival rate of 1 corresponds to the maximum number of events that can be handled by two nodes at the maximum voltage/frequency level. The average arrival rate of this trace is 26%. The other two curves are the power consumption for each PowerPC 750 system when there is no power management (NPM). From the plots, we can see that the power consumption of PowerPC 750 systems under NPM is mostly independent of event arrival rate (i.e., system load).

Figure 6.4b shows the power consumption for each PowerPC 750 system when LAOVS is used at the front-end. The power consumption of 0 corresponds to the node being powered down during that period. Figure 6.4b shows that using DVS in each node consumes much less power than not using any power management. However, we can also see that nodes may



a. Round-robin

b. LAOVS

Figure 6.4: The power consumption for PowerPC 750 systems (RR vs LAOVS)

be mistakenly powered down due to the inaccurate estimation of system load based only on the feedback of load information from each node (network communication delay is one of the factors).

Table 6.6: System energy consumption for RR and LAOVS, normalized to RR/NPM

Policies	Energy(kJ)	%
RR/NPM	37.7	100.0
RR/DVS	13.4	35.5
LAOVS/NPM	26.3	69.5
LAOVS/DVS	10.7	28.4

Table 6.6 shows the total energy consumption in the system when different policies are employed. We can see that only 69.5% energy was consumed by LAOVS compared to RR when NPM is employed on nodes. When DVS is used on each node, LAOVS uses around 20% less energy compared to RR.

6.3 HETEROGENEOUS REAL-TIME CLUSTERS

Global and local power management policies are next proposed for the general case of heterogeneous server clusters. For validation, we describe and evaluate an implementation of the

proposed schemes using the *Apache* Webserver in a small realistic cluster. This is the first power management scheme that is *simultaneously* (a) local and cluster-wide (i.e., turning on and off machines), (b) designed for heterogeneity, (c) QoS-aware and power-aware at the local servers (i.e., deadline-aware), (d) measurement-based (contrary to theoretical modeling, relying on measurements is the key to our approach), and (e) implementation-oriented.

Table 6.7 compares our scheme with prior work in the area of local and global energy management for servers and server clusters. For completeness, note that the local PM scheme in [27] combines DVS with request batching.

Table 6.7: Energy management for servers and server clusters

Paper	Local PM	Global PM	Cluster type	QoS-aware	Power	Implem.
Pinheiro et al. [64]	-	on/off	homogeneous	-	-	$\surd$
Heath et al. [40, 41]	-	on/off	heterogeneous	-	modeled	$\surd$
Sharma et al. [79]	DVS	-	-	$\surd$	-	$\surd$
Elnozahy et al. [26]	DVS	on/off	homogeneous	-	modeled	-
Bohrer et al. [10]	DVS	-	-	-	modeled	-
Elnozahy et al. [27]	DVS	-	-	-	modeled	-
Xu et al. [92, 93]	DVS	on/off	homogeneous	-	modeled	-
Proposed approach	DVS	on/off	heterogeneous	$\surd$	measured	$\surd$

6.3.1 Cluster Model

The cluster model is the one in Figure 6.3, with the mention that servers are heterogeneous. A front-end machine receives requests from clients and *redirects* them to a set of processing nodes, henceforth referred to as *servers*. The front-end is not a processing node and has three main functions: (a) accepting aperiodic requests from clients, (b) selecting the server to handle each request, and (c) reconfiguring the cluster (i.e., turning servers on/off) to reduce the global energy consumption while keeping the overall performance within a prespecified QoS requirement. After receiving a request, the front-end communicates to the client to which server the request must be sent using *HTTP redirection* [15]. Then, the client sends its request directly to the server.

In our cluster scheme, each request is an aperiodic task (i.e., no assumptions are made about task arrival times) and is assigned a deadline. The specification of the QoS is system-

wide and is, in our case, the percentage of deadlines met.

Each server in the heterogeneous cluster performs the same service (i.e., all servers can process all requests). No restriction is imposed regarding any aspect of their computation: process scheduling, CPU performance, memory speed/bandwidth, disk speed/bandwidth, power consumption, network bandwidth, etc. In addition, servers periodically inform the front-end about their current load, to aid the front-end in load distribution and cluster configuration decisions. After a request has been processed by a server, the result is returned directly to the client, without the front-end as intermediary.

Note that a more common cluster design is with the front-end acting as a proxy (i.e., acting as intermediary between clients and servers). In our webserver example, choosing one configuration or the other (i.e., proxy versus no proxy with redirection) is simply a configuration option, and the proposed scheme in this paper works equally well with either type of front-end. In our experiments, for high loads (above 1Gbps), we had to use the no-proxy architecture shown in Figure 6.3, as a proxy front-end cannot fully utilize the cluster in our experimental setup (our front-end has only one GbE network interface card).

6.3.2 Global Power Management

Our proposed front-end follows a very general framework that is applicable to *any* heterogeneous cluster. To achieve this goal, we cannot impose any restriction on server characteristics. However, for ease of presentation, definitions and examples emphasize web server clusters.

6.3.2.1 Load Definition and Estimation The front-end determines the number of active servers to meet the desired level of QoS while minimizing cluster energy consumption. The number of servers is computed (offline or online) as a function of the system load. Thus, defining load correctly is a crucial step. A measure of the load for clusters is the number of requests received per second, measured over some recent interval [79]. Clearly, depending on the kind of service under consideration, other definitions of load may be more appropriate (such as the bandwidth for a file server).

At runtime, the front-end needs to correctly estimate (or observe) the load, in order to

make PM decisions and to perform load distribution. The load estimation can be further improved by using feedback from the servers. In our case load is defined as described in Section 4.2 (Equation 4.3, that is the CPU load) and feedback is used only for preventing server overloading. Two types of requests are considered: static pages and dynamic pages.

The *maximum load* of a server is defined as the maximum number of requests that it can handle meeting the 95% of deadlines, which is our QoS requirement. The front-end never directs more than the maximum load to a server. The cluster load is defined as the sum of the current loads of all active servers. Therefore, the maximum load that the cluster can handle is the sum of the maximum loads of all servers. At runtime, the cluster load (i.e., the number of static pages N_{static} and dynamic pages $N_{dynamic}$) is observed every *monitor_period* seconds. The value of *monitor_period* is a design issue, related to the tradeoff between response time and overhead. In our cluster, values in the order of a few seconds were found suitable.

6.3.2.2 Server Information In order to reduce the global power consumption at runtime, we furnish the front-end (at startup) with information about the average power consumption of each server for any different value of its load. Servers can reduce their own power consumption in a number of different ways, such as using DVS and low-power states for the CPU, self-refresh modes for memory, stopping disk spinning after some time of idleness, etc. Moreover, each server may use a different OS or a different scheduling policy (such as a standard round robin, or a real-time policy to give higher priority to static pages with respect to dynamic ones). No assumption is made at the front-end about the local PM schemes (that is, the shape of the power functions).

The parameters for each machine are reported in Table 4.3 (on page 34). The information and corresponding notation about each server (needed at the front-end level) is presented next. $boot_time_i$ and $shutdown_time_i$ represent the time to boot and to shutdown server i, including the time to start and finish the (webserver) process of the server. max_load_i is the maximum load of server i that can satisfy the 95% QoS requirement. off_power_i is the power consumed when the server is off (some components, such as the Wake-On-LAN interface used to power up the machine, may not be completely off). Finally, $power_vs_load_i$ is an array with $\lceil \frac{max_load_i}{load_increment} \rceil$ entries recording the measured power consumption of server

i for each value of the load in $load_increment$ percents (we used 1%). The first entry of the array denotes the idle power (i.e., no load). $power_vs_load_i$ corresponds to the measured P_{load} power functions shown in Figure 4.3 (on page 37).

6.3.2.3 On/Off Policy The front-end, besides distributing the load to servers to minimize global power consumption, determines the cluster configuration by turning on/off servers. The front-end algorithm turns machines on and off in a specific order, which is based on the power efficiency of servers (i.e., the integral of power consumption versus load). In our case, according to the values of Figure 4.3, the ordering is: *Transmeta*, *Blue*, *Silver*, *Green*. In some situations we may need to change the order at runtime, as explained later.

The front-end turns on servers as the cluster load increases. However, since the boot time is not negligible, we need to turn machines on *before* they are actually needed. For this reason, the front-end maintains a variable, called $max_load_increase$, which specifies the maximum load variation that the cluster is prepared to sustain during $monitor_period$. This is essentially the maximum slope of the load characterization for the cluster.

The on/off policy relies on two tables computed offline. $mandatory_servers$ is a table that keeps the load at which to turn servers on and is used to determine the lowest number of servers needed at a certain load to satisfy the QoS requirement. For example, consider a cluster with three servers having maximum loads $max_load_0 = 0.5$, $max_load_1 = 1.5$ and $max_load_2 = 1.0$, respectively. Suppose that $monitor_period$ is 5 seconds, $max_load_increase$ is equal to 0.05, and the boot time is 10 seconds for every machine. Ideally, we need only one server when the cluster load is less than 0.5, two servers when load is between 0.5 and 2, and all servers when load is higher than 2. However, if we account for the time to boot a new machine and we suppose that the cluster load is checked periodically every $monitor_period$ seconds, the table becomes $mandatory_servers = \{0, 0.35, 1.85\}$. Thus, the first server is always on, whereas the second and third servers are turned on when the cluster load reaches 0.35 and 1.85, respectively. In fact, if we consider the boot time of a new server, we have to account for a potential load increase equal to $\frac{boot_time}{monitor_period} max_load_increase$. Moreover, if we suppose that the load is checked periodically every $monitor_period$ seconds, we have to introduce an additional interval of time to account for the error when measuring

the current load. In general, server i is turned on when the cluster load reaches

$$\sum_{j=0}^{i-1} max_load_j - (\frac{boot_time_i}{monitor_period} + 1) max_load_increase \quad (6.1)$$

The second table, called *power_servers*, precomputes the number of servers needed to minimize the power consumption for a given load. Unlike the previous table, this table is computed considering the power consumption of servers, and is used to distribute the current load among active servers. For a given value of N, we compute the power consumption of the cluster as follows. We start considering a load equal to zero, and we increase its value in *load_increment* increments. Each load increment is allocated to the server that increases its power by the least amount (according to its power function), thus minimizing the overall energy/power consumption. In other words, the server with the smallest power slope is allocated the current load increment. Note that as the load of a server increases, its power slope is expected to increase due to the convex relationship of power and voltage/frequency.

To determine the load at which N servers become more power efficient than $N-1$, we follow this procedure considering both cases of $N-1$ and N machines, respectively. The load at which N servers consume less power than $N-1$ servers is the value after which the Nth server is turned on. The server to be turned on is the next one according to the power efficiency order.

The complexity of computing the two tables is $O(N)$ (where N is the number of servers) for *mandatory_servers* and $O(NM)$ for *power_servers*, where $M = \sum_{i=1}^{N} \lceil \frac{max_load_i}{load_increment} \rceil$. In our cluster, the time to compute these two tables was less than $1msec$, which was negligible compared to *monitor_period* (which is in the range of seconds). Thus, this computation can also be performed online. For example, a new ordering of the servers and an online recalculation of the tables become necessary when a server crashes or when a server is upgraded.

A high-level view of the front-end on/off policy is presented in Algorithm 4. Every *monitor_period* seconds the load is estimated according to Equation 4.3, then the request counters are reset. The number of mandatory servers $N_{mandatory}$ is determined by a lookup in the *mandatory_servers* table. If $N_{mandatory}$ is higher than the current number of active servers $N_{current}$, all needed servers are immediately turned on.

Algorithm 4 On-off policy

1: Compute the load according to Equation 4.3
2: Reset the counters: $N_{static} = 0$ and $N_{dynamic} = 0$
3: Compute the minimum number of servers: $N_{mandatory}$=$mandatory_servers(Load)$
4: **if** $(N_{mandatory} > N_{current})$ **then**
5: turn on the servers, set $N_{current} = N_{mandatory}$,**return**
6: **end if**
7: Compute the low-power number of servers: $N_{power} = power_servers(Load)$
8: **if** $(N_{power} > N_{current})$ and $(Cmd \neq$ Boot$)$ **then**
9: Set $Cmd =$ Boot, find the next server i to boot
10: Set $N_{current} = N_{current} + 1$, **return**
11: **end if**
12: **if** $(N_{power} < N_{current})$ and $(Cmd \neq$ Shutdown$)$ **then**
13: Set $Cmd =$ Shutdown, find the next server i to shutdown
14: Set $N_{current} = N_{current} - 1$, **return**
15: **end if**
16: **if** Cmd=Boot for a period of time equal to $time_boot_i$ **then**
17: Turn on server i, set $Cmd =$ None, **return**
18: **end if**
19: **if** Cmd=Shutdown for a period of time equal to $time_boot_i + time_shutdown_i$ **then**
20: Turn off server i, set $Cmd =$ None, **return**
21: **end if**

Each server can be in one of the following states: Off, Boot, On, or Shutdown. After receiving the "boot" command (such as a Wake-On-LAN packet), the server i moves from the Off to the Boot state. It stays in this state for $boot_time_i$ seconds (i.e., until it starts the server process), then informs the front-end that it is available for processing, moving to the On state. When server i is shutdown, it stays in the Shutdown state for $shutdown_time_i$ seconds, after that the front-end changes its state to Off.

The variable Cmd in Algorithm 4 can have three different values: None, Boot or Shutdown. This variable allows to describe the use of thresholds when turning on/off servers. If no server is in transition (i.e., all servers are in the On or Off states) a server may be turned on or off, as decided after a lookup in the $power_servers$ table. To be conservative, only one server at a time is turned on or off. Server i is turned off if the system is in state Cmd = Shutdown for at least $time_shutdown_i + time_boot_i$ consecutive seconds, which is the rent-to-own threshold. Similarly, server i is turned on if Cmd = Boot for $time_boot_i$ consecutive seconds. Notice that these thresholds do not apply to the mandatory servers, which are started immediately. The running time of the online part of the algorithm (every $monitor_period$ seconds) is negligible because it is in the microsecond range; the complexity

is $O(N)$, but can be improved to $O(1)$ by increasing the table size from N to M (that is, storing all entries in an array).

For convex and linear power functions, tables *mandatory_servers* and *power_servers* contain the optimal transition points (in the discrete space; for continuous space, see [75]). In practice, however, power functions may have concave regions. This means that a server with an abrupt power increase at some load x may not be allocated more than x load, even though the power may become flat above $x + \epsilon$, making it a good target for load allocation. A simple fix to the problem is to consider the average power consumption over a larger interval, rather than the exact value at each load. This effectively results in smoothing the power functions. In our case, although the measured power functions have concave regions, we have found that no smoothing was necessary (as there are no abrupt power changes).

6.3.2.4 Request Distribution Policy The front-end distributes the incoming requests to a subset of the current servers that are in the On state. *load_allocation* is a table containing the estimated load allocated to each server and is computed with the same procedure used to determine the *power_servers* table, in $O(MN)$ time. The load allocation is computed every *monitor_period* seconds, after the on/off decisions.

Another table, called *load_accumulated*, stores the accumulated load of each server in the current period, and is reset after computing *load_allocation*. The server i with the minimum weight

$$w_i = \frac{load_accumulated_i}{load_allocation_i} \tag{6.2}$$

gets the next request. Notice that w_i can be higher than 1 when the cluster load (and thus $load_allocation_i$) was underestimated. The server that receives the request updates its accumulated load (and thus increases its weight), by adding $A_{static}/monitor_period$ or $A_{dynamic}/monitor_period$, depending on the request type (recall that A_{static} and $A_{dynamic}$ are the average execution times of static and dynamic requests respectively). The complexity of finding the server with minimum weight is $O(N)$ with a straightforward implementation, but can be improved to $O(logN)$ using a tree.

6.3.2.5 Implementation Issues We implemented our PM scheme in the *Apache* 1.3.33 Web server [2]. We created an Apache module, called *mod_on_off*, which makes on/off decisions. Moreover, we extended an existing module, *mod_backhand* [33], to support our distribution policy.

mod_backhand is a module responsible for load distribution in Apache clusters. It allows servers to exchange information about their current usage of resources. It also provides a set of *candidacy* functions to forward requests from an overloaded server to other less utilized servers. Examples of such functions are *byLoad*, which selects as candidate the least loaded server, and *byCost*, which considers a cost for each request.

We added a new candidacy function, called *byEnergy*, to implement our request distribution policy. We only enabled the *byEnergy* candidacy function in the front-end machine (that is, the servers do not redistribute requests). In addition, servers provide some feedback about their current real-time utilization to front-ends. We used this feedback to prevent the overloading of the servers. The server with the minimum w_i is selected, and w_i increases faster for overloaded servers. In more detail, we add twice the cost of a request (that is, A_{static} or $A_{dynamic}$) when updating w_i for an overloaded server (100% load or higher). We choose this approach of reducing the rate as opposed to temporarily suspending the sending of requests to overloaded servers due to practical limitations of *mod_backhand*: the load feedback occurs only several times per second. Thus, suspending the sending of requests to an apparently overloaded server results in unnecessary overloading of other servers.

The *mod_on_off* module communicates with *mod_backhand* through shared memory. On initialization, *mod_on_off* acquires server information and computes both *mandatory_servers* and *power_servers* tables. *mod_on_off* executes periodically every *monitor_period* seconds. On each invocation it performs the following tasks: (a) computes the current load based on the counters N_{static} and $N_{dynamic}$ (that are incremented in the Apache *post-read* request phase), (b) looks up in the table to determine the number of servers needed for the next period, (c) computes the *load_allocation* table for the active servers (not shown in Algorithm 4), (d) turns on (by sending Wake-On-LAN packets) and off (by invoking special CGI scripts) servers, and finally (e) resets the counters N_{static}, $N_{dynamic}$ and *load_accumulated*.

In addition, we modified the *mod_backhand* graphical interface to display at runtime the

estimated power and energy consumption of each server, based on the *power_vs_load* and *load_accumulated* tables. The interface is shown in Figure 6.5 and displays the current state and estimated power consumption of each server, as well as the total cluster energy since startup.

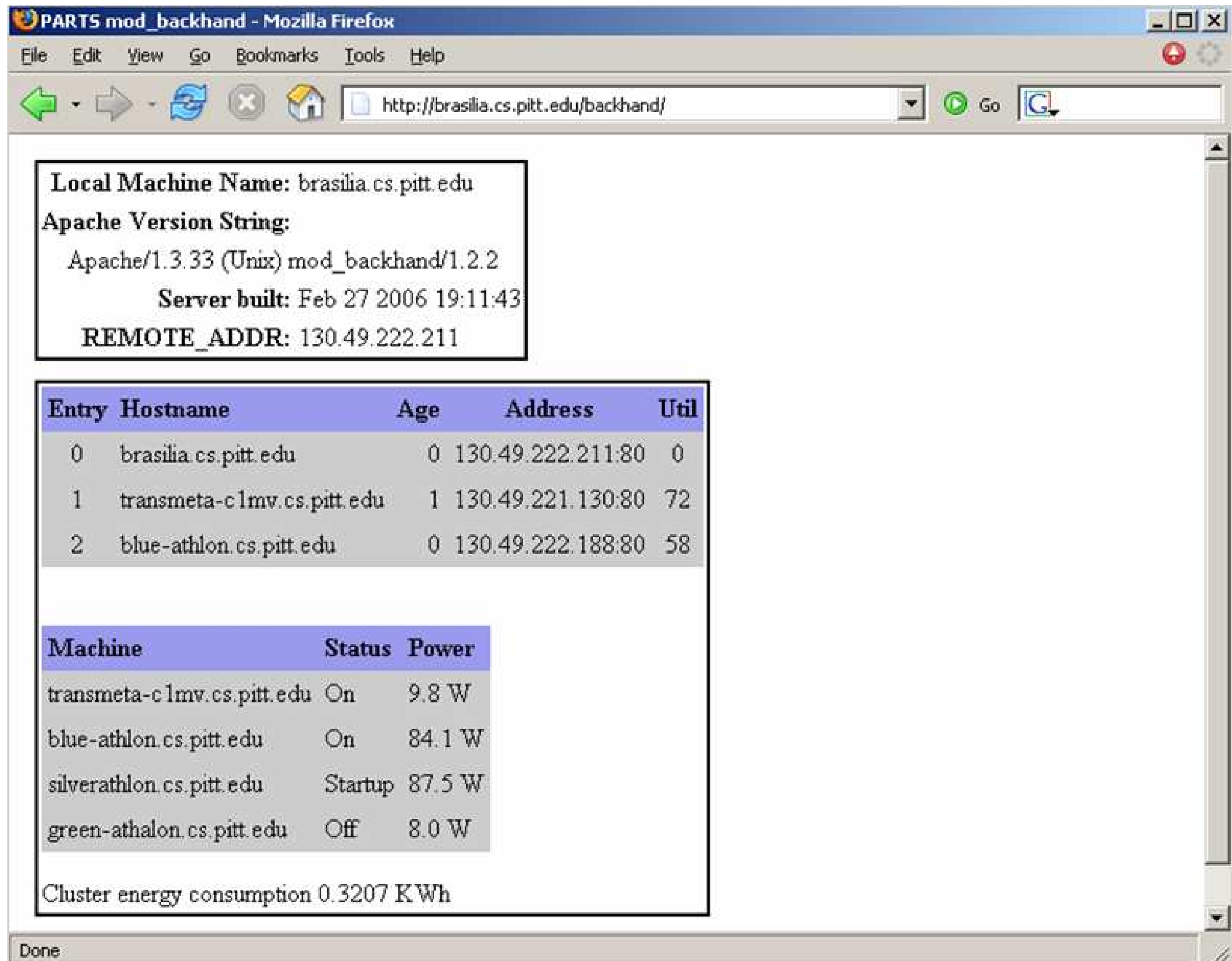


Figure 6.5: Online cluster power and energy estimation

6.3.3 Local Power Management

In addition to front-end directed cluster reconfigurations (i.e., turning on/off machines), the servers perform their own local PM to reduce power consumption of unutilized or under-utilized resources. We present an example of a QoS-aware DVS scheme and we discuss an

implementation using the Apache Webserver [2].

6.3.3.1 Local DVS Policy We rely on a local real-time scheme, where each request is an aperiodic task and is assigned a deadline. Each request *type* [73, 93, 41] has a deadline to allow for more accurate load estimation.

We consider a *soft* real-time system, in which the schedule is not generated by a real-time scheduler and the computation time C_i is the average execution time (i.e., A_{static} or $A_{dynamic}$), not the worst-case. Let D_i be the time remaining to the deadline, then the real-time utilization of a server is defined as $U = \sum_i \frac{C_i}{D_i}$.

If the CPU is the bottleneck of the system (as in our case), the CPU frequency to handle this rate of requests is Uf_M, where f_M is the highest possible frequency of the CPU. Each server periodically computes its utilization U and sets the CPU frequency to the closest value higher than Uf_M.

Note that DVS architectures may have inefficient operating frequencies [74], which exist when there are higher frequencies that consume less energy. A simple online tool for inefficient frequency elimination has been provided in [61]. Removal of inefficient operating frequencies is the first step in any DVS scheme. This was not necessary in our servers, because surprisingly all frequencies were efficient, although we had a different experience with other systems we tested [93].

6.3.3.2 Implementation Issues We implemented an Apache module, called *mod_cpufreq*, responsible for CPU speed settings at the user level. After detecting the available frequencies, our module creates an Apache process that periodically sets the CPU frequency according to the current value of U. We chose a small period of 10ms, but large enough so that the 50μs measured overhead of voltage/frequency changes (in the Athlon64 machines) is negligible.

To compute U, the module needs to know the type (i.e., static or dynamic) and the arrival time of each request. At every request arrival (called Apache *post-read* request phase), the arrival time and the deadline are recorded with μs accuracy and stored in a hash table in shared memory. Requests are removed from the queue after being served (called Apache *log* request phase). The request type is determined from the name of the requested file. Thus,

a single queue traversal is necessary to compute U. In fact, the current value of U depends on all queued requests, therefore the complexity is $O(R)$ where R is the number of requests queued; the overhead is negligible, as at most dozens of requests are queued at any time.

A problem we encountered during the implementation was that our scheme worked very well except for fast machines serving a large amount of small static pages. In this case, those machines were not increasing their speed, resulting in a large number of dropped requests. Further investigation revealed that the value of U was close to zero. We did not see this phenomenon on slower machines (such as Transmeta) nor on bigger requests. The problem was that the requests were served too fast (in approximately 150 μs). Such short requests were queued, served, and removed from the queue before other requests were added to the queue. Thus, at any time only a few requests (usually just one) was in the queue, and when *mod_cpufreq* recomputed the utilization, it resulted in an underestimation of U. In other words, even though the requests were received and queued at the OS-level, Apache was not able to see them because it is a user-level server and it has no information about requests stored at the OS level. We called this problem the *short request overload problem* phenomenon.

A simple fix was to compute the utilization also over a recent interval of time $interval_size$ (we used $200ms$):

$$U_{recent} = \frac{(N_{static}A_{static} + N_{dynamic}A_{dynamic})}{interval_size} \tag{6.3}$$

We would like to keep the server utilization U_{recent} below a certain threshold (we used $threshold = 80\%$). The minimum frequency that does that is $\frac{U_{recent}}{threshold}f_M$. Thus, our module sets the CPU speed to $max(U, \frac{U_{recent}}{threshold})f_M$. Note that Sharma et al.'s work with a kernel webserver (kHTTPd) [79] aware of small requests at the OS-level has a nice synergy with our approach and could be used in lieu of our scheme. The problem with including such work in our scheme is exactly the reason why development of kHTTPd was discontinued: the difficulty of maintaining, developing and debugging a kernel-level server. Exploring the composition of our cluster configuration and Sharma's (or other similar DVS) work is left for future work.

6.3.4 Evaluation

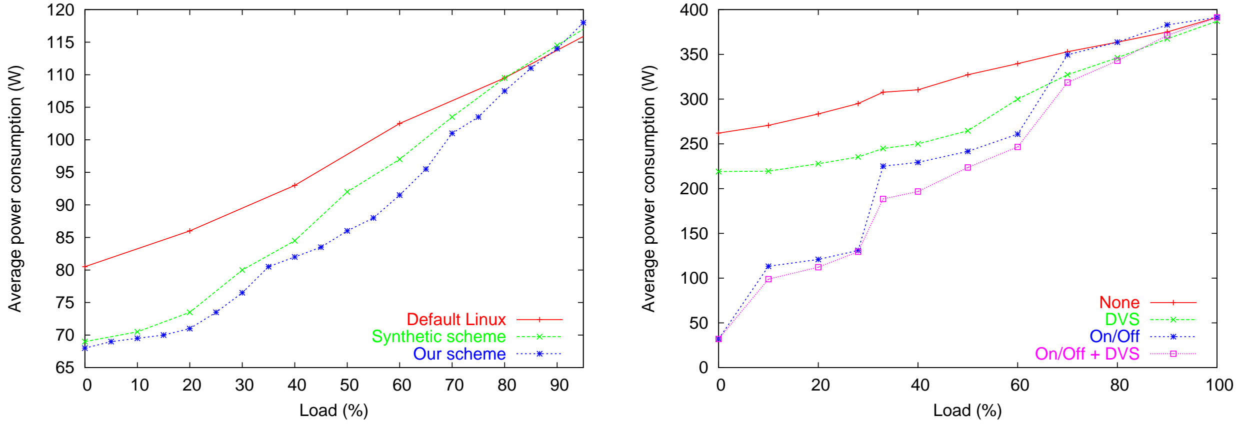
To evaluate our QoS-aware PM schemes we used a small cluster composed by one front-end and 4 different servers. Every machine ran *Gentoo Linux 2.6* as operating system and *Apache* 1.3.33 servers. The parameters of the machines are shown in Tables 4.3 and 4.4.

The cluster has been tested using 2 clients connected to the cluster with a GbE interface and Gbps switch; the clients generate up to 3,186 requests per second, which corresponds to a total maximum cluster load equal to 2.95 (all loads were normalized to that of *Silver* machine). A total cluster load of 0.05 (or 5%) corresponds on average to 54 requests/second. Considering request types, however, greatly improves the prediction, as 54 requests/second may correspond to a load ranging from 0.02 (if $N_{dynamic} = 0$) to 1.32 (if $N_{static} = 0$). We assigned deadlines of $50ms$ and $200ms$ for requests of static and dynamic pages, respectively.

We set $max_load_increase = 0.005$. Based on the measured server power functions, the following tables were derived: $power_servers = \{0, 0.1, 1.05, 2.04\}$ and $mandatory_servers = \{0, 0.062, 1.012, 2.012\}$.

6.3.4.1 DVS Policy As first experiment, we evaluated the effectiveness of our local DVS scheme. We compared our *mod_cpufreq* module with the default PM in Linux (i.e., HALT instruction when idle) and with Sharma's DVS scheme for QoS-aware web servers proposed in [79] (which we implemented at user level in our *mod_cpufreq* module). This scheme adjusts the speed of the processor to the minimum speed that maintains a quantity called *synthetic utilization* below a theoretically derived utilization bound. The bound that ensures that all deadlines are met is $U_{bound} = 58.6\%$ [1].

The measured power consumption of each scheme on the *Blue* machine is shown as function of the load in Figure 6.6a. The graph shows that our scheme outperforms the other schemes, especially for the mid-range load values. Higher savings are obtained on machines with a more convex power function (the power function of the *Blue* machine is rather linear, as seen in Figure 4.3). In fact, for a rate of 300 requests/sec (approximately 28% load) the average processor frequency is 1.25GHz using our scheme and 1.5GHz using Sharma's scheme, but the amount of energy saved is only 3%. Importantly, we observed that both



a. Comparison of DVS policies

b. Cluster power consumption

Figure 6.6: Evaluation of local and global power management techniques

schemes maintained the QoS level above 99% even at the highest load.

6.3.4.2 Overall Scheme To evaluate the overall scheme, we performed many experiments with and without the cluster-wide PM scheme (On/Off scheme), and with and without the local PM scheme (DVS scheme). For each load value, we measured the power consumption of the entire cluster for each scheme independently (see Figure 6.6b). For fairness, we used the load balancing policy in Section 6.3.2.4 for all the schemes.

The On/Off policy allows a striking reduction of the energy consumption for low values of the load, because it allows to turn off unutilized servers. In Figure 6.6b we can see that when load = 0, the cluster consumption is around 32W because each Athlon server consumes 8W when in the Off state, and the Transmeta also consumes 8W when in the On state. The DVS technique, instead, has its biggest impact whenever a new server is turned on, since not all active servers are fully utilized. However, its importance decreases as the utilization of the active servers increases. For high values of the load (in our case, at 70% or higher) all servers are on, therefore the On/Off technique does not allow to reduce energy consumption. In those situations, however, there is still room for the DVS technique, that becomes more important than the On/Off technique.

The energy consumption of all servers without any power management scheme was

1.32KWh. On average, we measured energy savings of 17% using DVS, 39% using On/Off, and 45% using both schemes. It is worth noting that the front-end estimation of the total energy consumed when using DVS was extremely accurate: the difference from the actual values was less than 1% in our experiments. For example, when using the on-off scheme, the measured value was 0.72KWh, while the front-end estimated value was 0.725KWh (the resolution of our power/energy meter [76] is 0.01KWh).

To measure the impact of cluster-wide and local PM schemes in the loss of QoS, we ran many four-hour experiments with workloads derived from actual webserver traces, and generated with the same shape of statistics taken from our *cs.pitt.edu* domain (see Table 4.5). The average delay (observed at the client side) without any PM scheme was 8.29ms; the small response time is due to all machines being on at all times, and running at maximum frequency. Adding DVS (local PM) had a very small impact on the delay, with the average delay measured at 8.77ms. However, with On/Off scheme, we measured an average delay equal to 12.29ms without DVS and 12.83ms with DVS. In both cases, the average delay was not higher than 50% of the no-PM delay and was quite small with respect to deadlines.

6.4 CHAPTER SUMMARY

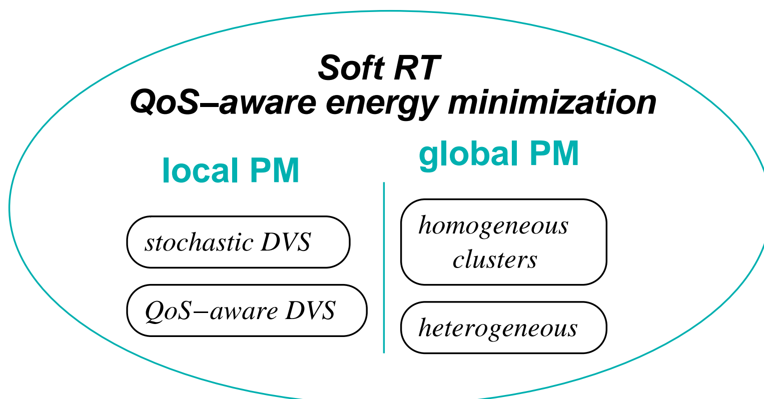


Figure 6.7: Chapter overview: QoS-aware energy minimization for soft real-time systems

The chapter summary is shown in Figure 6.7. The focus is on QoS-aware energy minimization schemes for soft real-time systems, corresponding to the second part of the research

overview in Figure 3.2.

We first investigate DVS algorithms for unpredictable requests corresponding to aperiodic soft real-time tasks. A stochastic algorithm is shown to largely outperform both prediction-based and utilization-based DVS algorithms, for realistic signal-processing application traces provided by our research partners. As the traces were shown to have a large variability in request execution times, prediction-based DVS schemes generally fail to accurately describe the load, while utilization schemes are either too aggressive or too slow to react to load changes. The stochastic scheme eliminates prediction and relies entirely on the probability distribution function of execution times to gradually increase the frequency/voltage of tasks aware of deadline constraints. The scheme results in 50% extra savings (on average) over the second-best policy for synthetic traces, up to 40% savings for real signal-processing application traces, and an order of magnitude less energy versus no power management.

For systems with more predictable behavior (such as web servers), a DVS scheme that combines prediction with utilization is shown to successfully reduce the energy consumption while guaranteeing a required QoS. The workload is predictable in this case, as web requests are typically small and numerous. Thus, using average execution times and inter-arrival times gives an accurate load estimation, even though variation in request execution times may exist. The use of request types is shown to further improve prediction.

We then direct our investigation to QoS-aware power management of multi-processor systems (clusters). We start with a homogeneous cluster model and propose a coordinated load-aware global (on/off) and local (DVS) policy (LAOVS) based on estimating the overall cluster load. The policy is shown to outperform previously proposed schemes that rely on ideal power models and ignore server idle periods. We evaluate our policy with an implementation in a prototype signal-processing embedded cluster and show that as much as 70% saving can be achieved compared to no power management.

Finally, we extend our work to the general case of heterogeneous clusters. A coordinated local/global policy is proposed that uses utilization-based DVS locally for each server and a global scheme to turn on/off servers as required by the cluster load. Both local and global power management schemes are QoS-aware. In addition, the global scheme relies on measurements (as opposed to theoretical models) to distribute requests in a power-aware

fashion. Consequently, it is shown that the estimation of cluster energy consumption is within 1% of real measurements. To prove the concept we implement the scheme in an experimental Apache web server cluster and show that 45% savings are achieved on average for real traces.

7.0 CONCLUSIONS

While performance is traditionally the main concern in systems design, we are now seeing a paradigm shift from a maximum performance approach to energy-performance tradeoffs. This is especially true in two areas of computing: embedded systems and high-performance computing. In the embedded arena, device lifetime is limited by the energy stored in a battery. In high-end servers, energy is a major concern for different reasons: energy costs and expensive cooling techniques. Thus, the way power and energy is managed plays an important role in the design of these systems.

Systems today provide a variety of mechanisms for power management. Examples of such mechanisms include dynamic voltage scaling (DVS) for processors, low-power states (such as self-refresh or power-down) for memory chips, or stopping disk spinning during periods of inactivity. However, power management does not come for free and should be used carefully. The use of low-power mechanisms has a negative impact on performance, due to the various overheads involved. A power management (PM) mechanism should only degrade performance within specified constraints.

Real-time systems provide a framework for specifying such performance constraints. In a hard real-time system all applications must finish by a specified deadline. A soft real-time system allows for occasional deadline misses but requires some kind of statistical guarantees. In addition, some application domains are adaptive, and application reward (or value, or utility) depends on the amount of resources allocated to the application. System reward/utility is then defined as the sum of rewards of all applications. For non-adaptive applications, the system reward corresponds to the quality of service (QoS) provided to users. Analyzing the issues of reward and energy aware of task deadlines is the focus of this work.

Specifically, we are investigating two types of energy-reward interplay for real-time sys-

tems: (a) system reward maximization mechanisms for hard real-time systems running adaptive applications with tight energy constraints, and (b) energy minimization for soft real-time systems running non-adaptive applications with QoS constraints.

We first explore reward maximization techniques for periodic and frame-based real-time application models. Assuming continuous frequencies/voltages and continuous application rewards, we first prove a set of properties of the optimal solutions for reward maximization. The properties are used to develop exact solutions for certain types of power functions. For the general case of power functions, an iterative algorithm is proposed. We validate the accuracy of the algorithm through simulations, and show that, on average, the system reward is within 1% of the optimal.

Extending to the more realistic case of discrete power and reward functions, we propose two algorithms for reward maximization: *REW-Pack* and *REW-Unpack*. The algorithms are shown to have running times in the microsecond range, which makes them suitable for dynamic environments with rapidly changing task sets and system requirements. Although searching for solutions from opposite directions (corresponding to tight energy and tight timing constraints respectively) the algorithms are shown to return very similar results, proving that energy and timing constraints are equally important. In our experiments we find that the system rewards returned by the algorithms are on average within 3% from the optimal, with higher percentages for task sets with tighter energy and timing constraints.

The discrete reward model is extended to incorporate task versions, corresponding to adaptive applications. The *REW-Pack* algorithm is augmented with the improved reward model into *MV-Pack*. The *MV-Pack* algorithm thus identifies not only the tasks selected for execution and their frequency/voltage (as was the case with *REW-Pack*) but also a task version for each task. As with *REW-Pack* and *REW-Unpack*, we show through simulations that *MV-Pack* has small average errors and running times.

While the previous algorithms for both continuous and discrete models make best use of the available energy in a relatively short period of time (associated with frame boundaries or the least common multiple of task periods) we also addressed the problem of long-term reward maximization. Specifically, we investigated long-term energy allocation policies for battery-powered systems and systems that rely on rechargeable energy. For rechargeable systems,

considering worst-case charging/discharging scenarios and worst-case application execution times, theoretical conditions are derived for system stability (that is, a task schedule is guaranteed to exist). Three dynamic policies are then proposed to further improve the system reward by reclaiming unused energy when worst-case conditions do not happen. While the static policy is too conservative and wastes energy, the dynamic policies generally take advantage of all scavenged energy and consistently improve the long-term system reward.

For soft real-time systems corresponding to aperiodic tasks we investigate energy minimization algorithms that maintain a required QoS. We first propose a stochastic DVS algorithm for unpredictable workloads. The algorithm relies on the probability distribution function of task execution times to gradually increase the frequency/voltage of tasks aware of deadlines. The algorithm is shown to outperform prediction-based and utilization-based schemes, which are not good choices for unpredictable workloads. 50% energy savings are reported for synthetic traces and up to 40% for real signal-processing application traces over the second-best DVS scheme. For workloads with more predictable behavior, such as web servers with a large (and predictable) number of relatively short requests, a utilization-based scheme is proposed that considers QoS and request deadlines. The scheme is implemented at user level in the Apache web server and shown to outperform default power management (that halts an idle processor) as well as a previously proposed kernel-level utilization-based DVS scheme.

We then direct our investigation to multiprocessor systems, in particular server clusters. Extending on previous work for homogeneous clusters we propose a load-aware global (on/off) and local (DVS) scheme (LAOVS). The scheme turns on and off servers as required by the overall cluster load and uses DVS for local power management independently at each server. We also eliminate the dependency on theoretical power models and consider the cluster load for on/off decisions. Accordingly, LAOVS is shown to outperform previously proposed schemes that turn on/off machines based on the average frequency in the cluster. LAOVS was implemented in an experimental embedded cluster and shown to result in 3.5x reduction over no power management (noPM) for realistic signal-processing traces.

For the general case of heterogeneous clusters, we propose an integrated local/global power management scheme that relies on determining the power functions of individual

servers through offline power measurements. Requests are allocated to servers considering primarily the measured power functions, as opposed to server performance. The scheme proposed is general and can apply to a variety of systems. As a proof of concept we implemented the scheme in an experimental Apache web server cluster. For realistic web traces we show savings of 45% on average over noPM, and up to 90% for reduced workloads. The scheme achieves the desired QoS without dropping requests and maintains request average response times well below their deadlines.

As energy estimation is crucial to this work, we also propose two energy/power estimation schemes. First, a model-based processor power estimation scheme is shown to be within 6% of measured values for a PPC405GP embedded processor across 39 embedded benchmarks. The model relies on certain energy events whose values are determined through experimentation. A second scheme was designed for estimating the system power consumption of servers, as a function of server load. This approach relies on realistic traces and offline power measurements. Using the measured power functions in a realistic web server cluster, the cluster total energy was estimated within 1% of real measurements.

To summarize, the contributions of my doctoral work to the state of the art in energy management and reward maximization are as follows:

- Theoretical properties are identified for reward maximization problems and an iterative system reward maximization algorithm is derived for continuous power and reward models. The algorithm is the first to consider energy, deadline and rewards simultaneously [69].
- A new discrete reward model is proposed for adaptive applications that do not reward partial execution: task versions [70]. The model is realistic and corresponds to discrete completion points, as well as different task parameters (corresponding to a variety of existing multimedia applications), tasks with different algorithms with various accuracies, or simply different invocation periods.
- Three efficient reward-maximization DVS algorithms are proposed for the realistic case of discrete rewards and power functions [68, 70]. The algorithms start with a relaxed schedule that is adapted according to various metrics considering task rewards, deadlines and energy consumptions.

- Theoretical results and a static energy allocation policy are identified for long-term reward maximization. Three dynamic policies are proposed to further improve the overall system reward by reclaiming unused energy [71, 72].
- A stochastic DVS scheme is developed for soft real-time systems, and is shown to largely outperform prediction-based and utilization-based DVS schemes for unpredictable workloads [73].
- Local (DVS) and global (on/off) schemes are combined in the context of soft real time homogeneous server clusters. The approach is based on evaluating the system load and was implemented in a prototype satellite-based embedded cluster [92, 93].
- A power management policy is proposed for the general case of heterogeneous server clusters. The policy is based on offline power measurements and is the first work that combines global and local techniques for heterogeneous clusters. We implemented and evaluated the scheme on a real Apache web server cluster [67].

BIBLIOGRAPHY

[1] T. F. Abdelzaher and C. Lu. Schedulability Analysis and Utilization Bounds for Highly Scalable Real-Time Services. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, Taiwan, June 2001.

[2] Apache. HTTP Server Project. http://httpd.apache.org/.

[3] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, Delft, Netherlands, June 2001.

[4] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the 22nd Real-Time Systems Symposium (RTSS'01)*, London, UK, 2001.

[5] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130, February 2001.

[6] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 113–121, March 2003.

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

[8] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, 2004.

[9] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo - a full system simulator for the powerpc architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), 2004.

[10] P. Bohrer, M. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. *The case for power management in web servers.* Kluwer Academic Publishers, 2002.

[11] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[12] H. Bryhni, E. Klovning, and O. Kure. A Comparison of Load Balancing Techniques for Scalable Web Servers. In *IEEE Networks*, pages 58–64, July 2000.

[13] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.

[14] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, December 1998.

[15] V. Cardellini, M. Colajanni, and P. S. Yu. Redirection algorithms for load sharing in distributed web-server systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, Austin, TX, June 1999.

[16] A. Chandra, W. Gong, and P. Shenoy. An online optimization-based technique for dynamic resource allocation in gps servers. Technical Report TR02-30, Department of Computer Science, University of Massachusetts Amherst, 2002.

[17] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuit*, 27(4):473–484, 1992.

[18] E. Chang and A. Zakhor. Scalable video coding using 3-d subband velocity coding and multi-rate quantization. *IEEE Int. Conf. On Acoustics, Speech and Signal Processing*, July 1993.

[19] J.-Y. Chung, J. W.-S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 19(9):1156–1173, September 1993.

[20] Embedded Microprocessor Benchmark Consortium. Eembc technical and operating specification. http://www.eembc.org.

[21] Transmeta Corporation. Longrun technology. http://www.transmeta.com/crusoe/longrun.html.

[22] C. Curescu and S. N. Tehrani. Time-aware utility-based resource allocation in wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):624–636, July 2005.

[23] J. K. Dey, J. Kurose, and D. Towsley. On-line scheduling policies for a class of iris (increasing reward with increasing service) real-time tasks. *IEEE Transactions on Computers*, 45(7):802–813, July 1996.

[24] J. K. Dey, J. Kurose, D. Towsley, C. M. Krishna, and M. Girkar. Efficient on-line processor scheduling for a class of iris (increasing reward with increasing service) real-time tasks. In *Proceedings of ACM SIGMETRICS Conference on Measurement an Modeling of Computer Systems*, pages 217–228, May 1993.

[25] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification. Second Edition*. John Wiley and Sons, New York, 2001.

[26] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Workshop on Power-Aware Computer Systems (PACS'02)*, 2002.

[27] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *4th USENIX Symposium on Internet Technologies and Systems*, Seattle, March 2003.

[28] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'97)*, pages 598–604, October 1997.

[29] X. Fan, C. Ellis, and A. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems (PACS'03)*, December 2003.

[30] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *International Conference on Supercomputing (ICS'05)*, pages 293–302, Cambridge, Massachusetts, June 2005.

[31] W. Feng and J. W.-S. Liu. An extended imprecise computation model for time-constrained speech processing and generation. In *Proceedings of the IEEE Workshop on Real-Time Applications*, May 1993.

[32] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. In *Proceeding of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[33] Center for Networking and Distributed Systems at Johns Hopkins University. The Backhand Project. http://www.backhand.org/.

[34] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Mobile Computing and Networking*, pages 13–25, 1995.

[35] J. Grass and S. Zilberstein. A value-driven system for autonomous information gathering. *Journal of Intelligent Information Systems*, 14(5-27), March 2000.

[36] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *International Symposium on Low Power Electronics and Design*, Aug. 2001.

[37] B. D. Guenther. Aided and automatic target recognition based upon sensory inputs from image forming systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):1004–1019, 1997.

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

[39] T. Heath, A. P. Centeno, P. George, Y. Jaluria, and R. Bianchini. Mercury and freon: Temperature emulation and management in server systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, October 2006.

[40] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Self-Configuring Heterogeneous Server Clusters. In *Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, September 2003.

[41] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[42] I. Hong, D. Kirovski, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *Design Automation Conference*, 1998.

[43] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Computer-Aided Design (ICCAD'98)*, pages 653–656, 1998.

[44] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings of the 19th IEEE Real-Time systems Symposium (RTSS'98)*, Madrid, Spain, December 1998.

[45] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design*, pages 197–202, Aug. 1998.

[46] C. M. Krishna and K. G. Shin. *Real-time Systems*. Mc Graw-Hill, New York, 1997.

[47] C.M. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 156–165, June 2000.

[48] P. Kumar and M. Srivastava. Predictive strategies for low-power rtos scheduling. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Austin, TX, September 2000.

[49] R. Lencevicius and A. Ran. Can fixed priority scheduling work in practice? In *Proceedings of the 24th IEEE Real-Time systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003.

[50] P. Li, B. Ravindran, and E. D. Jensen. Adaptive time-critical resource management using time/utility functions: Past, present, and future. In *28th International Computer Software and Applications Conference (COMPSAC)*, September 2004.

[51] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of ACM*, 20(1):46–61, March 1973.

[52] J. Liu, P. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints for mission-critical embedded systems. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, Las Vegas, NV, June 2001.

[53] J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[54] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithms with pace. In *ACM SIGMETRICS*, June 2001.

[55] Y.-H. Lu and G. De Micheli. Adaptive hard disk power management on personal computers. In *Proceedings of the IEEE Great Lakes Symposium on VLSI*, pages 50–53, March 1999.

[56] D. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley, Reading Massachusets, 1984.

[57] Intel Xscale Microarchitecture. http://developer.intel.com/design/intelxscale/benchmarks.htm.

[58] A. Miyoshi, C. Lefurgy, E. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, New York, June 2002.

[59] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, Philadelphia, PA, October 2000.

[60] A. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.

[61] PARTS. Power Efficiency Test. http://www.cs.pitt.edu/PARTS/demos/efficient.

[62] PARTS. Power-sim. http://www.cs.pitt.edu/PARTS/PACC/NEW.

[63] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th Symposium on Operating Systems Principles*, October 2001.

[64] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, Sept 2001.

[65] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, December 1997.

[66] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for qos-based resource allocation problems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[67] C. Rusu, A. Ferreira, C. Scordino, A. Watson, R. Melhem, and D. Mossé. Energy-efficient real-time heterogeneous server clusters. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 418–427, San Jose, California, April 2006.

[68] C. Rusu, R. Melhem, and D. Mossé. Maximizing the system value while satisfying time and energy constraints. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Austin, December 2002.

[69] C. Rusu, R. Melhem, and D. Mossé. Maximizing rewards for real-time applications with energy constraints. *ACM Transactions on Embedded Computing Systems*, 2(4):1–23, 2003.

[70] C. Rusu, R. Melhem, and D. Mossé. Maximizing the system value while satisfying time and energy constraints. *IBM Journal of Research and Development*, 47(5/6):689, 2003.

[71] C. Rusu, R. Melhem, and D. Mossé. Multi-version scheduling in rechargeable energy-aware real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, July 2003.

[72] C. Rusu, R. Melhem, and D. Mossé. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing*, 2004.

[73] C. Rusu, R. Xu, R. Melhem, and D. Mossé. Energy-Efficient Policies for Request-Driven Soft Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, July 2004.

[74] S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.

[75] C. Scordino and E. Bini. Optimal Speed Assignment for Probabilistic Execution Times. In 2nd *Workshop on Power-Aware Real-Time Computing (PARC'05)*, NJ, September 2005.

[76] Seasonic. Power Angel. http://www.seasonicusa.com/power_angel.htm.

[77] H. Shafi, P. Bohrer, J. Phelan, and C. Rusu. Event-based power simulation. In *IBM Austin Conference on Energy-Efficient Design (ACEED)*, Austin, TX, February 2002.

[78] H. Shafi, P. Bohrer, J. Phelan, C. Rusu, and J. L. Peterson. Design and validation of a system performance and power simulator. *IBM Journal of Research and Development*, 47(5/6):641, 2003.

[79] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Liu. Power-aware QoS Management in Web Servers. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003.

[80] D. C. Sharp, E. Pla, and K. R. Luecke. Evaluating mission critical large-scale embedded system performance in real-time java. In *Proceedings of the 24th IEEE Real-Time systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003.

[81] W.-K. Shih, J. W.-S. Liu, and J.-Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal on Computing*, 20(3):537–552, 1991.

[82] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design and Test of Computers*, pages 18(23):20–30, March 2001.

[83] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th Design Automation Conference (DAC)*, 1999.

[84] P. M. Shriver, M. B. Gokhale, S. D. Briles, D. Kang, M. Cai, K. McCabe, S. P. Krago, and J. Suh. A power-aware, satellite-based parallel signal processing scheme. *Power Aware Computing*, pages 243–259, 2002.

[85] J. L. Stone. Photovoltaics: Unlimited electrical energy from the sun.

[86] C. J. Turner and L. L. Peterson. Image transfer: An end-to-end design. In *SIGCOMM Symposium on Communications Architectures and Protocols*, August 1992.

[87] S. V. Vrbsky and J. W. S. Liu. Approximate - a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(12):1056–1068, December 1993.

[88] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, California, 1994.

[89] C. C. Wust, L. Steffens, R. J. Bril, and W. F Verhaegh. Qos control strategies for high-quality video processing. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, Sicily, Italy, June 2004.

[90] C. C. Wust, L. Steffens, and W. F. Verhaegh. Adaptive qos control for real-time video processing. In *Proceedings of the Work-in-Progress Session of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.

[91] M. Xiong, S. Han, and K.-Y. Lam. A deferrable scheduling algorithm for real-time transactions maintaining data freshness. In *IEEE Real-Time System Symposium*, Miami, Florida, December 2005.

[92] R. Xu, C. Rusu, D. Zhu, D. Mossé, and R. Melhem. Practical energy-efficient policies for embedded clusters. In *6th Brazilian Workshop on Real-Time Systems(WTR '04)*, Gramado, Rio Grande do Sul, Brazil, May 2004.

[93] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé. Energy-Efficient Policies for Embedded Clusters. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, Chicago, IL, June 2005.

[94] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995.

[95] 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 (SOSP'03)*, Bolton Landing, NY, October 2003.

[96] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *International Conference on Computer Aided Design (ICCAD'04)*, pages 35–40, San Jose, CA, November 2004.