ABSTRACT

Title of Thesis: RTOS-BASED DYNAMIC VOLTAGE SCALING

Degree candidate: Nuengwong Tuaycharoen

Degree and year: Master of Science, 2003

Thesis directed by: Professor Bruce L. Jacob
Department of Electrical and Computer Engineering

Energy consumption of real-time embedded systems becomes more important as such systems are widely used in many applications. In those systems, the core processor consumes a large amount of the total energy. Dynamic voltage scaling (DVS) is accepted as the key technique to reduce energy dissipation of the core by lowering the supply voltage and operating frequency. Currently, most DVS heuristics are based on average values of the past utilization, either with or without real-time constraint guarantees. This thesis is aimed at exploring the possibility to apply a classical control-systems technique, namely PID controller, to a DVS heuristic. The PID controller is able to find a good average value that represents the past, the present, and the changing workload of the system. By applying the characteristics of real-time application

programs, the technique can also meet the real-time constraints. The technique is integrated into μC/OS II, a multitasking preemptive real-time operating system running on a Motorola M-CORE processor model. The applications used in the experiments are all members of the MediaBench benchmark suite. The experimental results show that the technique can reduce significantly energy consumption by consuming only 5% above the PERFECT case, and consuming only half of the energy consumed by the AVGN algorithm. Besides, the technique also guarantees the real-time constraints by preserving both jitters and miss-deadline rates below 5%.

RTOS-BASED DYNAMIC VOLTAGE SCALING

by

Nuengwong Tuaycharoen

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2003

Advisory Committee:

Professor Bruce L. Jacob, Chair
Professor Gang Qu
Professor David B. Stewart

ACKNOWLEDGEMENT

First of all, I would like to thank my advisor, Dr. Jacob for his dedication and encouragement. He has helped me develop the idea and data interpretation. Without his guidance, this thesis would have never been possible.

I would like to thank Brinda and Ankush who are always there for me and handed me this wonderful idea. I will never find the way out without these gorgeous people. Also, I would like to thank Dr. Stewart and his students, Tom and Nitin, who set up the motivated reading seminar in Real-Time Systems and also all students attending the class. The reading seminar had a great influence in the development of the thesis.

Finally, I would to thank my family, my father and mother who are always be the great mentors, my aunt and her family who always be my second family while I am away from home. Especially, thank to Matin who always encourage and give me any assistance for this thesis as many times as I ask.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Real-Time Embedded Systems

Embedded systems have become increasingly popular in industrial applications and in consumer markets. Such systems are used, for example, in automotive applications and cellular phones. Embedded systems contain processors running specific software while, in traditional electro-mechanical systems, most functionality is implemented in hardware. Embedded systems leave only computation intensive functions as well as system parts that require fast response times or high concurrency to the dedicated system hardware to reduce the cost per unit since they are commonly produced in large amounts.

Most embedded processors are based on architectures designed primarily for general-purpose processors. Both kinds demand massive data processing and high performance. However, there are design factors differentiating the embedded processors from the general-purpose processors: power consumption, code density, cost, and integrated peripherals [1].

Power consumption plays an ever-increasingly-important role since embedded processors are widely used in portable and inaccessible systems, operated by battery. Maximizing the battery life is one of the key concerns because the amount of energy available to these systems is limited. Power

consumption has to be considered along with performance. Reducing power consumption almost always means reducing the processor performance. With this requirement, increasing overall system performance will be a major challenge for future embedded-processor designs.

The number one reason in selecting an embedded operating system is the real-time capabilities of the operating system [2]. The fact is the majority of the embedded systems market has been traditionally made up by real-time systems, which are characterized by deterministic, low-latency performances. Thus, the correct behavior of these systems depends not only on the accuracy of computations but also on their timeliness. Such systems have typically been used in aerospace, military, industrial automation, telecommunications and automotive applications, as well as in other safety-critical applications that require deterministic response times. While energy consumption for embedded and mobile computing is important, energy minimization must be carefully balanced against the need for real-time behaviors.

Recently, the synthesis of application-specific systems in embedded systems has changed drastically due to a convergence of applications, technology, and market trends. Market trends are moving towards growing the number of gates in the integrated circuits, and shortening the clock period. Also, embedded consumer applications are growing in size of code. Qualitatively, modern

applications demand high volume of data processing, cost efficiency, and very short time-to-market windows [3].

By adopting design reuse techniques, real-time operating systems (RTOSs) can narrow down the gap between the fast development in semiconductor technology and the embedded system designs. RTOS isolates application developers to architecture and integrated circuits, plus the processor architects and the circuit designers do not have to be concerned with the applications. As a result, the percentage of developers using no RTOS of a proprietary alternative has decreased from 38% to 18% in five years, and the percentage of those using RTOS is increasing significantly [4].

1.2 Dynamic Voltage Scaling

The known fact is the core processor consumes a large portion of energy [5]. Therefore, reducing the energy consumed by the core is not trivial. By trading performance for energy savings, one can employ these three methods [6],

1) Voltage Scaling,
2) Transistor Sizing, and
3) Adiabatic Circuits.

Recent technologies allow the implementation of a microprocessor system that can adjust the operating voltage and the clock frequency at run-time. The dynamic voltage scaling (DVS) decreases voltage to an appropriate level

whenever it is possible to cut energy consumption. The voltage scheduler in the system RTOS makes the decision when and which level to scale.

Voltage scheduling can be separated into two tasks[7]:

- Load Prediction: predicting the future system load based on past behavior.
- Speed-Setting: using the load prediction to set the voltage level and clock frequency.

Dynamic voltage scaling (DVS) is accepted as the key technique to reduce energy dissipation by lowering the supply voltage and operating frequency in response to the concept of performance on demand (see Figure 1). Lowering only the operating frequency can reduce the power consumption, but the energy consumption remains the same because the computation needs more time to finish. Lowering the operating frequency and also supply voltage accordingly can reduce a significant amount of energy because of the quadratic relation between the energy and the voltage supply.

There are two categories in the voltage schedulers [5]. First, interval-based voltage schedulers simply analyze system utilization at a global level: no direct knowledge of individual threads or programs is needed. However, it has a disadvantage of incorrectly predicting future workloads. As a result, it fails to guarantee real-time constraints.

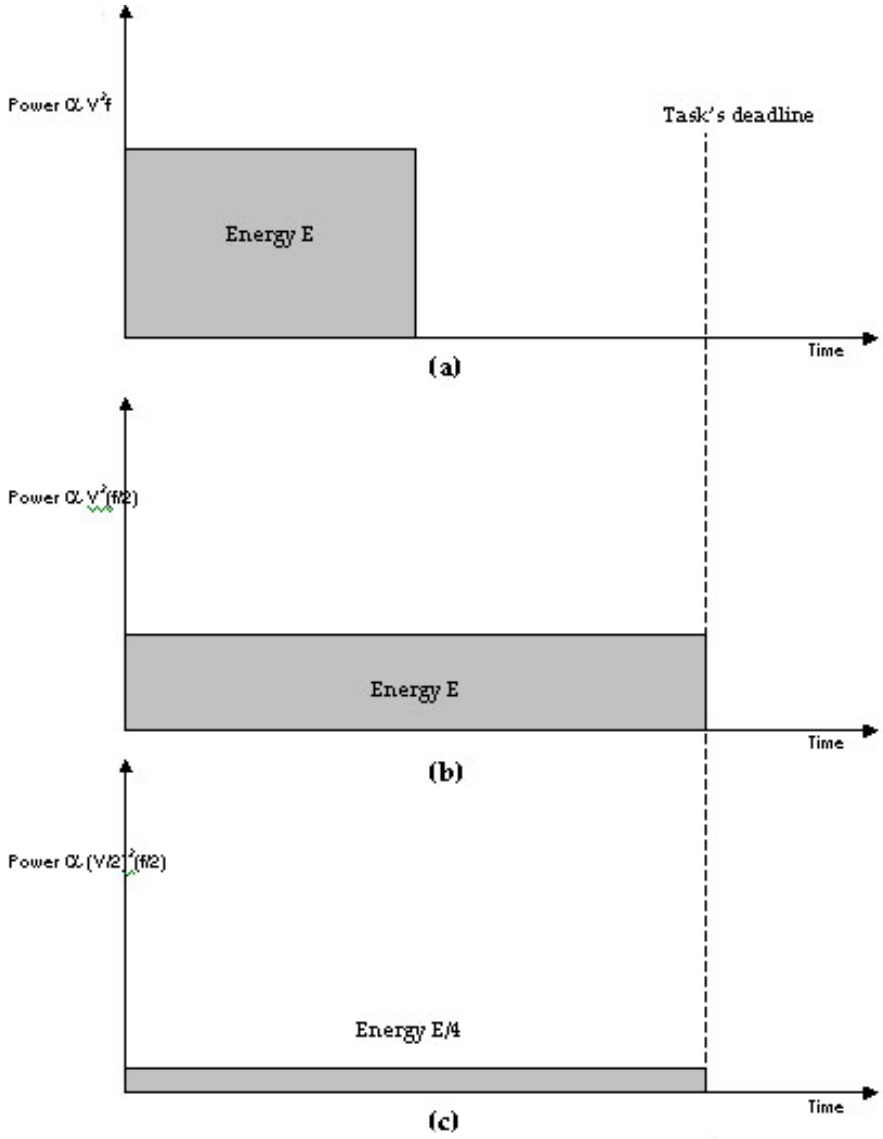

Figure 1. Energy Consumption vs. Power Consumption. (a) Task ready at time 0; no other task is ready. Task requires time T to complete, assuming maximum clock frequency f. (b) Reducing the clock frequency f by half lowers the processor's power consumption and still allows task to complete by deadline. However, it only spreads the computation out overtime. The energy consumption remains the same. (c) Reducing the voltage level V by half reduces the power level further, without any corresponding increase in execution time. As a result, the energy consumption is reduced significantly, but the appropriate performance is remained.

The second category is thread-based voltage schedulers, which require knowledge of individual thread deadlines and computation. These schedulers can calculate optimized speeds according to the execution requirements and deadlines specified by each thread. This type of scheduler is also subdivided into inter-task schedulers, where the voltage is changed on task-by-task basis, and intra-task schedulers, where the voltage is changed within the individual task boundary [8].

There are only three key components for implementing DVS in a general microprocessor system:

1) an operating system that can intelligently vary the processor speed,
2) a regulation loop that can generate the minimum voltage required for the desired speed, and
3) a microprocessor that can operate over a wide voltage range.

Therefore, DVS is possible in many systems [9-11].

Currently, most DVS heuristics are based on the average value of the past system utilization, both with and without considering real-time responsiveness. Most studies concentrate on a particular application such as MPEG decoders and turbo code decoders, which have wide variation in their workload. The latest work explores the possibility to identify more slack time and efficiently spreading the workload over it.

1.3 Motivation

This thesis aims at designing a DVS heuristic integrated into an RTOS, which can satisfy the following requirements.

- It should be transparent to user programs.
- It can work well with the multiprogrammed systems.
- It guarantees real-time constraints for all hard-real time tasks.
- It has little computation compared to user program.
- It is independent of the RTOS task scheduler.
- It leads to settle to an optimum operating point fast and remains stable in case of periodic tasks.

A classical control-systems technique, namely Proportional-Integral-Derivative controller (PID), is applied to find an appropriate value that represents the past, the present, and the changing workload of the system. Ishihara [12] brought up interesting theories about the number of voltage levels that maximize energy saving. The theories show that, in order to produce a schedule that minimizes energy consumption, only one or at most two voltage levels, which are the neighbors of the optimum voltage, are needed. This implies that an algorithm aimed at producing a sole optimum speed value, could be an optimized algorithm. In addition, they concluded that the processor that has more voltage levels tends to be able to minimize the energy consumption. Therefore, the future

trend in microprocessors is moving toward a processor that can operate in relatively continuous range of the voltage levels. The algorithm having its complexity independent of the number of operating voltage levels should be promising. Furthermore, by applying the real-time behavior of the applications, the technique can also meet the real-time constraints.

1.4 Overview

This paper is organized as follows. The following chapter discusses the background, i.e. the energy reduction in CMOS, Real-time systems, Real-time operating systems and the task scheduling, the PID controller, and the recent DVS works. Chapter 3 gives the details of the proposed heuristic and chapter 4 gives the experiment-setup details for this project. Chapter 5 presents the experiment results and analysis. Finally, chapter 6 ends the thesis with a conclusion statement.

CHAPTER 2

BACKGROUND

2.1 Energy Reduction in CMOS

The energy required for a computation on a single static CMOS device is [11]:

$$E = C_{tot} V_{dd}^2 + I_{leak} V_{dd} \left(\frac{N}{f}\right)$$

Or, simply,

$$E \propto V_{dd}^2$$

with the parameters defined as follows:

- C_{tot} is the total switched capacitance for the computation,
- N is the number of clock cycles taken by the computation,
- V_{dd} is the supply voltage,
- f is the clock frequency, and
- I_{leak} is the leakage current.

A reduction in supply voltage increases the circuit delays as shown by the following equation:

$$T_d = \frac{C_L V_{dd}}{\mu C_{ox}(W/L)\,(V_{dd} - V_t)^2}$$

or, simply,

$$f \propto \frac{(V_{dd} - V_t)^2}{V_{dd}}$$

where

T_d is the delay or the reciprocal of the frequency f,

V_{dd} is the supply voltage,

C_L is the total node capacitance,

μ is the mobility,

C_{ox} is the oxide capacitance,

V_t is the threshold voltage, and

W/L is the width to length ratio of transistors.

The variations in processor utilization affects N and C_{tot}. Then, due to a light workload, an idle processor wastes clock cycles and energy because the clock signal continues propagating, and the operating voltage remains the same. Gating the clock during idle cycles reduces the switched capacitance of idle cycles. Reducing the frequency f during periods of low workload eliminates most idle cycles altogether. However, reducing the frequency f is limited by V_t, which must be large enough to overcome noise in the circuit. Neither approaches, however, affects $C_{tot}V_{dd}^2$ for the actual computation or substantially reduces the energy lost to leakage current. Reducing the supply voltage V_{dd} in conjunction with the frequency f achieves energy savings and reduces leakage current.

Dynamic voltage scaling (DVS) is the on-line adjustment of V_{dd} and f in response to a processor's utilization. A voltage scheduler, running in addition to an operating system's task scheduler, can adjust voltage and frequency in response to a prior knowledge or predictions of the system's workload.

2.2 Real-Time Systems

The *Oxford Dictionary of Computing* offers the definition of the word *Real Time* [13]:

"Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in

the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."

The most important characteristic features of the real-time systems is not only the correctness of its functional computation (logical correctness), but also the correctness of timely response (timely correctness). In the real-time world, the fastest response does not mean the best performance. However, the system must guarantee its timely response to external events.

Since one can trade one feature for another, the real-time can be broadly categorized into *hard real-time systems* and *soft real-time systems*. In the hard real-time systems, timing correctness is critically important and may not be sacrificed for other gains. On the other hand, in soft real-time systems, the timing correctness is important but not critical. An occasional failure can be acceptable. One can view the concept of soft real-time as a continuous spectrum with non-real-time at one end-point, and hard real-time as the other end.

A *job* is a unit of work that is scheduled and executed by the system. A set of related jobs that jointly provide some system function is a *task*. The *release time* of a job is the instant of time at which the job becomes available for execution. The *response time* is the length of the time from the release time of the job to the instant when it completes.

The timely correctness of a real-time system can be specified in the form of *period* and *deadline*. The *period* is the minimum amount of time between each iteration of a regularly repeated task. Such repeated tasks are called periodic tasks. The *deadline* is a constraint on the instant of time by which the operation must complete. Most of the time, for a job in a periodic task, a job's deadline is that job's period. Therefore, a hard real-time task has a *hard deadline*, and a soft real-time task has a *soft deadline*.

For periodic tasks, the least common multiple of the periods of all tasks is call *hyperperiod*. The *phase* of a task is the release time of the first job in that task. The utilization of a task is equal to the fraction of time a truly periodic task with period p_i and execution time e_i keeps a processor busy, or simply the ratio of e_i/p_i. Hence, the *total utilization U* of all tasks in the system is the sum of the utilization of every task. Note that the total utilization specifies how busy the processor is.

On the other hand, an *aperiodic task* is a task having either soft deadlines or no deadlines, while a *sporadic task* is released at random time and has hard deadlines.

2.3 RTOSs and Task Scheduling

ComputerWorld [14] gives a definition of the real-time operating system as follows:

"A real-time operating system (RTOS) is specialized control software that's often used in embedded computing applications that have tight memory resources and stringent performance requirements relating to immediate response times, high availability and accurate self-monitoring capabilities."

The most important characteristic of any RTOS is the ability to preserve the real-time behavior, namely, the correctness of both the functional computation and the timely response. The RTOS must also have high predictability.

A real-time operating system can supply many functions to an embedded application, including process management, interprocess communication and synchronization, memory management, and input/output (I/O) management. However, the central purpose of an RTOS is task scheduling, the mechanism making a decision which job in the system's job pool should execute at the instant of time. The approaches commonly used in task scheduling are [15]:

1) Clock-Driven Approach: the scheduler makes decisions at regularly spaced time instants.
2) Weighted Round-Robin Approach: a job with weight *wt* gets *wt* time slices every round, and the length of a round is equal to the sum of the weights of all the ready jobs.
3) Priority-Driven Approach: the scheduler makes a decision every time a job releases or completes.

The last approach can be categorized by the moment the scheduler makes decisions into preemptive and nonpreemptive scheduling. The scheduler of preemptive systems makes a decision every time a job releases, but the scheduler of nonpreemptive systems makes a decision at the time a job completes.

Priority-driven schedulers differ from each other in job priority assignment. A fixed-priority algorithm assigns a task priority and then remains the priority of that task for every job in the task. A dynamic-priority algorithm assigns different priorities to the individual jobs in each task. Therefore, with respect to other tasks, the priority of a task changes every time a job releases or completes.

In 1973, Liu [16] presented the *rate monotonic (RM)* algorithm as an optimal fixed priority scheduling algorithm, and the *earliest-deadline-first (EDF)* algorithms as optimal dynamic priority scheduling algorithms. The schedulers are *optimal* in the sense that they can generate a task schedule if it exists. These algorithms are widely used in today's operating systems. The reasons are not only the ease in implementation for the algorithms, but also the ease in verification. One can verify whether there is a feasible schedule for a particular periodic task set such that all the tasks' deadlines can be guaranteed by checking the schedulability conditions [15],

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq 1$$

, for EDF, and

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq n(2^{1/n} - 1)$$

for RM. These conditions are also called *acceptance test* [46], as they are used to check if a new arrival task can be scheduled to meet its deadline and all tasks already scheduled are still guaranteed to meet their deadlines. Note that the utilization of the schedulable system is only less than 70% for RM scheduling and 100% for EDF scheduling, when the number of tasks in the critical task set approaches infinity. As a result, [17] introduced maximum-urgency-first (MUF) algorithm to increase the bound utilization to somewhat over 100%.

2.4 PID Controller

A Proportional-Integral-Derivative (PID) controller is often used in control systems [15, 18]. To make the value of one variable follow the value of another with minimum error, the controller ensures that when the controlling variable

changes its value, the controlled variable changes accordingly. Usually, this involves keeping track of past values of both controlling and controlled variables. In a typical PID controller, these elements are driven by a combination of the system commands and the feedback signal from the object that is being controlled (usually referred to as the "plant"). Their outputs are added together to form the system output. A full PID control algorithm with feedback has the following equation:

$$y(t) = K_P x + K_I \int_0^t (x - y)\,dt + K_D \frac{dx}{dt}$$

in which y is the output of the controller at time t, and x is the input at time t.

- The Proportional part of the equation ($K_P x$) makes sure the system reacts as soon as there is a change in the input, and the change in new output tries to follow the input. Increasing the gain K_P results in the ability of the system to correctly track the current value of the input, and responds to it fast. However, high K_P may cause the output oscillated.
- The Integral part of the equation ($K_I \int_0^t (x - y)\,dt$) is referred to as the integral of the error of the feedback term, since it measures the net difference between the output and the input thus far. Applied with the proportional term, it

provides the stability under changes in the load on the process or some environmental condition by remembering the error in the past, keeping the compensated system stable. That is if the steady-state error to a given input is constant, the integral term reduces it to zero.

- The Derivative part of the equation ($K_D \frac{dx}{dt}$) makes sure that the system responds to sudden disturbance in the input efficiently. Since derivative term considers the rate of change of the error signal, it *anticipates* the future value and hence acts to reduce the error that would other wise arise from the disturbance.

Simpler control schemes often use just PI or PD controllers for efficient control. However, the PID controller is the one used where efficiency, stability and performance are all required. In fact, because the PID controller can cope perfectly with 90% of all control problems, it provides a strong deterrent to the adoption of new control system design techniques [13]. The next chapter will show how an adaptation of this kind of function can be used for efficient voltage scaling in today's microprocessors.

2.5 Related Works

At the beginning, the research in DVS mostly involved in interval schedulers [5] using the global system state to set the speed accordingly without

knowledge of the running user programs, i.e. the algorithms proposed by Govil, Weiser, Pering, and Chandrasena [7, 19-21]. First, the PAST algorithm was introduced by Weiser [19] et al. The algorithm calculated the utilization over the previous interval and assigned the new speed value accordingly. Expectedly, the algorithm may produce oscillated speed setting because it makes the decision upon the utilization that is changed by the previous speed itself.

Govil et al studied a wide range of heuristics from weighted averages of past behavior to pattern matching of processor utilization [7]. They concluded that the simple algorithms based on rational smoothing rather than "smart" predicting may be most effective. From their experiments, no patterns of the processor usage were found.

Pering et al [20] investigated a number of algorithms, i.e. PAST, COPT, and AVGN, over PDA benchmarks. The result showed that the performance of the algorithms depends on parameter setting and it differs over different benchmarks.

Grunwald's paper [22] evaluated a number of the algorithms proposed by Govil, Weiser, and Pering [7, 19, 20] by implementing them on a real system, Itsy, an experimental pocket computer that ran a version of LINUX. They concluded that no heuristics provides satisfied results by approaching the ideal speed for specific set of benchmarks they used. Specifically, they indicated that the best of

class, Pering's AVGN algorithm, could not settle on the clock speed that maximizes CPU utilization.

Chandrasena and Liebelt [21] proposed AQRS algorithm, which is only the average of the sampling workload, and then assign the speed or rate according to the average and the most recent workloads. They also suggested that the voltage dithering technique, which handles the optimal rate that does not equal to quantized rates by assigning to two different rates—one after another, significantly improves energy savings over the algorithms that simply set the voltage to the greater one.

More recent works concentrated on thread-based voltage schedulers that considered the individual thread or task parameters, such as deadline, execution time, and release time, in order to maintain timing constraints and minimize energy consumption at the same time. However, most of them still employ weighted average utilization over the past as the key idea , e.g. the algorithms proposed by Yao, Pering, Son, and Leung [23-26]. Another approach was to stretch the workload over available slack time to the next task's release time, e.g. the algorithms proposed by Shin, Leung, Pillai, Kim, Aydin, Chakrabarty, and Swaminathan [27-33].

Yao et al proposed a model of job scheduling aimed at energy minimization in EDF manner [23]. The scheduling algorithms, one for off-line and

one for on-line decision (AVR), are based on the knowledge of the exact execution cycles of every task in any scheduling interval.

Pering et al proposed another thread-based algorithm that set the processor speed based on empirically estimated workload and deadline of each thread [24]. The voltage scheduler set the speed according to the equation,

$$\mathrm{speed} = \underset{\forall(i \le n)}{\mathrm{MAX}} \left\lceil \frac{\sum_{j \le i} \mathrm{work}_j}{\mathrm{deadline}_i - \mathrm{currenttime}} \right\rceil$$

, while the work of task j is calculated by an exponential moving average at completion of an application frame. The scheduler is called every time a thread is added, removed, or reached deadline, in addition to periodically evaluation. The algorithm assumed to work with EDF scheduling.

They also claimed that the overhead of the scheduler is quite small such that it requires a negligible amount of throughput and energy consumption.

Shin and Choi [27] investigated using processor's power-down mode in conjunction with DVS, the processor reduced its voltage level when the operating systems detects there is only one active task in the run queue. The speed is set in such a way that the job's worst case execution time spreads until the release time of the first job located in the delay queue. On the other hand, the processor enters

the power-down mode if the operating system detects no active jobs, and the run queue is empty. The processor stays in the power-down mode until the next release time of the first job located in the delay queue.

Three heuristics were brought up by Pillai and Shin [29] working with RM or EDF task scheduling. The first one, static scheduling, selects only one lowest possible operating frequency to let all tasks meet all the deadlines. The second one, cycle-conserving scheduling, determines the lowest frequency for each schedule task satisfying the acceptance test. In the acceptance test, the bound of the total utilization is decreased to the optimum speed of the system. That is

$$U_1 + U_2 + \ldots + U_n \leq \frac{f_1}{f_{max}}$$

Then, the system updates the actual utilization, according to the full speed, that the previous task used in order to calculate the next task speed. The last heuristic, look-ahead scheduling, tries to spread all tasks backwards and considers the future tasks simultaneously. The simulation results showed that the look-ahead scheduling is the best among three heuristics in almost all cases.

Swaminathan [32, 33] introduced a task scheduling that decides the optimal voltage at the same time. The scheduler, LEDF, selects a task with EDF policy then tries to reduce the frequency. If all tasks still meet their deadlines with that scheduled task running with low frequency, the system reduces the

frequency for the scheduled task. However, this algorithm requires polynomial transformation to the period of the task set to the hyperperiod via LCM theorem.

While some works focused on DVS in general real-time systems, others focused on some specific applications, such as MPEG decoder [25], turbo code decoder [29, 34], and sensor network [35]. Son et al [25] focused on a heuristic for an MPEG application. The proposed heuristic employs the weighted average to find the decode time per byte in each interval. It is very similar to PID controller technique without proportional term. Therefore, a PID-based algorithm could give good heuristic in MPEG application.

Chung [36] focused on DVS over MPEG decoding application. The technique constructs necessary information, i.e. exact execution time while the VDO is encoding at the server site. Then, the MPEG file is sent with the information in order to be decoded at the client site.

Leung et al [26, 28] concentrated on the turbo code decoder. The algorithms aimed at maximizing the delay of each iteration. Their as-slow-as-possible (ASAP) algorithm decides the delay of each iteration by assuming that the later iterations need only minimal time (or maximum speed). They improved the ASAP algorithm by using the SNR. Another proposed algorithm is called History-Based Assignment Algorithm. It only averages the number of iterations

over n previous frames to predict the next one. Then, the scheduler calculates the optimal delay for the next frame from the iteration number.

Gilbert et al also focused on a heuristic for turbo-decoder in [37], which is based on the predicted number of current iteration on the average of the number of the iterations over the past 3 frames. Then, the processor speed is set accordingly.

Yuan and Qu focused on the application for the sensor network [35]. The proposed method embedded necessary information in the header of all messages produced. There are three voltage levels required for three functions, which are decryption, message processing, and encryption. The embedded information tells the system how much iteration it needs for each function. As a result, the system can select the appropriate voltage at the appropriate time.

Another interesting approach is to apply some probability models to the timing parameters of the tasks as the algorithms proposed by Sinha [38] and Simunic [39]. Sinha's algorithm [38] is a reduced-complexity form of the analysis of the relation between the optimum speed and utilization via probability, assuming known execution cycles and deadline of the task a stated. However, in their probability model, which is a binomial distribution, they assumed each time unit can be independently selected. While, in the real system, the previous time unit should be selected before the later unit. They also proposed a theory stating,

aside from the deadline, minimum energy consumption occurs when all tasks have the same averaged processing speed.

Simunic [39] et al extended the Dynamic power management (DPM) with their DVS algorithm. The DVS algorithm analyzed the stochastic model, assuming exponential distribution for both execution time and task arrival time, and queuing theory for prediction of execution time for MP3 and MPEG. The disadvantage of the algorithm was it required off-line analysis to set the P_{max} value, the maximum probability if the system used the maximum frequency. They compared its performance with Pering's algorithm [24].

Kim et al compared a diversity of algorithms in a preemptive, periodic, hard real-time system scheduled with EDF or RM [8]. They classified algorithms into 3 classes: interDVS which the frequency changes only between tasks, intraDVS which the frequency changes in a task, and hybridDVS which is the mixing between those two. Their simulation results, over SimDVS environment, showed that the heuristics laEDF in [29], lpSHE in [30], and DRA and AGR in [31] can give very good performance in energy consumption, in other word, very close to the optimal lower bound. lpSHE, DRA, and AGR analyzed slack times of other tasks and then effectively use them. The most important point was to identify all slack times in the system, and spread the current task over.

Qu [40] classified the dynamic voltage scaling systems into 4 classes:

1) Ideal which can vary the voltage arbitrary and can change immediately,
2) Feasible which can vary the voltage arbitrary between v_{min} and v_{max} while the system still continue executing,
3) Practical which stops executing during voltage changing, and
4) Multiple which has only a number of discrete supply voltages available and the system can transit from one level to another instantaneously.

The paper also gave the upper bound for energy saving for each class of the DVS systems.

Additionally, DVS is also applied to multiprocessor systems. Bambha et al [41] discussed a DVS scheduler in multiprocessor environment in static (off-line) approach. Their algorithm used the period graph method to find the solution locally, and the simulated annealing method to find the global solution between PEs.

Luo and Jha proposed a DVS heuristic in distributed real-time systems that can manage all periodic, aperiodic and sporadic tasks [42]. The heuristic assigns a static task schedule and voltage schedule to the periodic tasks, and reserves an appropriate amount of time for sporadic tasks. For the aperiodic tasks, it assumes Poisson distribution for the arrival time. Then, predict the next

interarrival time from the average arrival time in the past. Next, it estimates the available time to the task execution,

Available_time = min(predicted_next_arrival,

k-> latest_finish)-currenttime

And, the worst-case remaining execution time,

Worst_remaining = worst_exec_time-executed

Finally, the ratio of available_time to worst_remaining is referred to as the scaling speed.

Jha also gave a survey in current DVS and DPM in [43].

Both Lee [44] and Azevedo [45] focused on intra-task off-line techniques. Lee et al introduced an algorithm applying HFSM-SDF theory to determine the exact remaining workload of the task [44]. This approach constructs a graph to identify the exact execution path and then assign the speed accordingly. On the other hand, Azevedo et al introduced a novel intra-task DVS technique under compiler control that used program checkpoints [45].

Liu and Mok [52] defined two functions, the available cycle function (ACF) and the required cycle function (RCF) for the system workload. Their algorithm was generated by defining the DVS problem as a nonlinear optimization problem.

AbouGhazaleh et al [53] introduced an approach using the compiler and operating system corroboration that used fine-grained information about the execution times of a real-time application to reduce energy consumption. The compiler generates the power management hints (PMHs), which is the specific information for each source code's path. The speed scaling at every power management point (PMP) is based on the PMH information.

To our knowledge, only small number of works focused on the uniprocessor real-time systems having periodic, sporadic, and aperiodic tasks, i.e. the work introduced by Hong [46] and Flautner [47]. Flautner's work [47] categorized the tasks into periodic producers, periodic consumers, and interactive episodes. For each episode, the operating system extracts the necessary information from the system. Also, the proper deadline of the interactive episode is determined by the perception threshold, which is set by human-computer interaction literature. For periodic episodes, the scheduler stretches the episode's execution to the beginning of the next episode or to the beginning of the associated consumer episode. The interesting point is, for each type of the episodes, the scheduler sets the speed independently. As a result, the performance factor, or the speed in our term, does not show the smoothness. However, Flautner's algorithm did not concentrate on the real-time behavior of the applications. The later works of this research group concentrated on applying

this algorithm with other improvements in physical devices [34, 48] to reduce more overall energy of the system.

Hong's HPASTS algorithm [46] employed the acceptance test [15] to guarantee the real-time constraints for the system that has both periodic and sporadic tasks. For the speed setting, they only set the speed to the calculated utilization from the acceptance test. In this thesis, we compare our proposed RT-nqPID algorithm with the HPASTS algorithm.

CHAPTER 3

METHODOLOGY

A classical control-systems technique, namely PID controller, is applied to find an appropriate value that represents the past, the present, and the changing workload of the system. The adaptation of the controller, called RT-nqPID, inherits the tracking and stable properties from the original controller. The complexity of the algorithm is also independent of the number of operating voltage levels. Additionally, by considering the real-time behavior of the applications, the technique can also meet the real-time constraints.

The RT-nqPID algorithm is proposed in this thesis to solve the DVS problem. The RT-nqPID is integrated into an RTOS, but, due to the property of an interval scheduler, it is independent of the RTOS task scheduler. It is transparent to user programs in the multiprogrammed real-time systems. It has little computation compared to the user programs. As shown in chapter 5, it settles to an optimum operating point fast and remains stable in case of periodic tasks.

The proposed RT-nqPID algorithm consists of two parts:

1. The nqPID function whose job is to predict the workload.
2. The user program utilization part for real-time constraints guarantee.

3.1 The nqPID Function

From the PID controller in chapter 2, we need to simplify and make extensive changes to tailor it to the task of voltage scaling before we can implement it as a software algorithm executing on a digital computer. We would need to make the following changes:

- We would need to convert the equation from continuous-time to discrete-time, replacing the integral with a summation.
- Similarly, we would replace the derivative with the difference between the current value of x and the previous value of x. For simplicity, we employ the difference between the values in only one interval to represent the linear behavior of the system. The difference is considered as the first order derivative.
- We would remove the term y from the right hand side and thereby remove the feedback loop. Systems without feedback have simpler behavior than systems with feedback. Because our goal was to perform a first order exploration of control-systems theory to the task of dynamic voltage scaling, we felt this was an appropriate step. Systems containing feedback loops are much more complex than systems without feedback loops. However, they often provide better performance.

- We cut the summation in the remaining integral term (now an average) from an infinite series of terms to a finite series of terms. It now represents the average value of x over the past m intervals.
- We define utilization (also called the system load) as the fraction of cycles that are busy in a given interval, and workload as the product of utilization and CPU speed as the following definition:

$$\texttt{utilization} = \frac{\texttt{busyCycles}}{\texttt{busyCycles} + \texttt{idleCycles}} = \frac{\texttt{busyCycles}}{\texttt{totalCycles}}$$

$$\texttt{speed} = \frac{\texttt{currentCPUFrequency}}{\texttt{maximumCPUFrequency}}$$

$$\begin{aligned}\texttt{workload} &= \frac{\texttt{busyCyclesAtCurrentFrequency}}{\texttt{totalCyclesAtMaximumFrequency}} \\ &= \frac{\texttt{busyCyclesAtCurrentFrequency}}{\texttt{totalCyclesAtCurrentFrequency}} \\ &\times \frac{\texttt{currentFrequency}}{\texttt{maximumFrequency}} \\ &= \texttt{utilization} \times \texttt{speed}\end{aligned}$$

Applying these changes to the PID controller yields the following discrete-time equation, where the y term are the predicted workloads and the x terms are the measured workloads. The nqPID equation is,

$$y_{n+1} = K_P \times x_n + K_I \times \sum_{n-m+1}^{n} \frac{x_i}{m} + K_D \times (x_n - x_{n-1})$$

This equation represents a function that uses the previous m values of the workload to predict what the next value of the workload will be. This estimate of the workload is used to set the processor's speed and voltage level for the next time quantum. Throughout this paper, we will refer to the modified version of the PID controller for voltage scaling based on the above equation with "nqPID" with stand for "not quite PID". The pseudo-code for the simplified, discrete-time algorithm is given in Figure 2.

Note, we removed both the direct feedback loop and an indirect feedback loop from the original PID controller. The direct feedback loop is described as the y term in the controller. On the other hand, the indirect feedback loop is the effect caused by the relation between the paired speed and the utilization. Therefore, unlike the PID controller, the "nqPID" algorithm is not expected to demonstrate the oscillation behavior due to the bad choice of the coefficients.

The beauty of the nqPID is that it produces a good representative for the past, the present, and the change of the system workload. Instead of using system utilization to predict the future operating frequency, the workload-approach nqPID also settles to an optimum operating point fast and preserves smoothness. An algorithm that sets the speed according to only the system utilization directly may fail to stabilize at an optimum speed because the system utilization will

change with the speed, which is set according to the utilization, itself. However, as we will show in the next chapter, using only the nqPID method cannot prevent missing deadlines, while the system utilization is equal to 100%. The state of the system, where the utilization is equal to 100%, but the speed and the workload are lower than maximum, is called a "saturated state". As a result, even though the speed is not set to the maximum speed, the saturated state causes missing deadlines.

```
// window of previous N actual system loads
load_t load[WINDOW_SIZE];

// Proportional term
estimated_load = Kp * load[0];

//Integral term
for (j=0; tmp=0; j< WINDOW_SIZE; j++){
    tmp += load[j] / WINDOW_SIZE;
}
estimated_load += Ki * tmp;

//Derivative term
estimated_load += Kd * (load[0] – load[1]);

// Speed Setting
setPercentSpeed = estimated_load / MAX_LOAD;
```

Figure 2. Psuedo-code for the nqPID algorithm. The code shows only the speed-setting portion of the algorithm; as a result of the calculations, the CPU's speed is set to be directly proportional to the estimated load. Not shown is the update of the load array, any error-checking, etc.

3.2 The User Program Utilization

To prevent the saturated state described above, the nqPID function requires more “headroom”, the range of the speed that the function will settle, to find the optimal speed. The more the headroom there is, the more the range between the lowest selected speed and the highest selected speed. Since we assume the energy consumed by changing speed step is varied by the number of steps changed, we prefer narrow speed range, and, consequently, small headroom that will not cause system saturation.

As shown in the next chapter, the nqPID function exhibits that the headroom constant should make the heuristic selects the value speed to the optimum value. Hong et al [46] suggested that the optimal speed of the system be equal to the sum of the ratio between the task execution time and deadline, which is the same concept as the utilization in the acceptance test. Therefore, the proposed RT-nqPID heuristic consists of four terms,

$$y_{n+1} = K_P \times x_n + K_I \times \sum_{n-m+1}^{n} \frac{x_i}{m} + K_D \times (x_n - x_{n-1}) + K_U \times \sum_{i=0}^{\#tasks} \frac{C_i}{D_i}$$

where

K_U is an appropriate constant,

C_i is the worse-case execution time of task i,

D_i is the deadline of task i,

#tasks is the number of the running tasks,

and other variables are as described in the previous section.

The pseudo-code for the RT-nqPID algorithm is in Figure 3. It simply adds the headroom constant to ensure that the algorithm will never lead the system to the saturated state.

```
// window of previous N actual system loads
load_t load[WINDOW_SIZE];

// Proportional term
estimated_load = Kp * load[0];

//Integral term
for (j=0; tmp=0; j< WINDOW_SIZE; j++){
     tmp += load[j] / WINDOW_SIZE;
}
estimated_load += Ki * tmp;

//Derivative term
estimated_load += Kd * (load[0] – load[1]);

//Headroom constant
for (j=0; j< NUMBER_OF_TASKS; j++){
     u += task[j].worst_case_exe_time / task [j].period;
}
estimate_load += Ku* u;

// Speed Setting
setPercentSpeed = estimated_load / MAX_LOAD;
```

Figure 3. Psuedo-code for the RT-nqPID algorithm. The algorithm added the 'headroom' constant, which is only a multiple of the total utilization of the system.

3.3 The Comparisons of PID, nqPID, and RT-nqPID Equations

Compared with the PID controller equation is Chapter 2, each term of the nqPID and RT-nqPID algorithms can be interpreted as following:

- The Proportional part of the equation (the first term) predicts that the next value of the load will be the same as that seen the last time like the PAST algorithm [19]. It ensures that the system can react quickly to the most recent workload. If the workload changes suddenly, the operating system can react to it appropriately in the next interval.
- The Integral part of the equation (the second term) predicts that the next workload will be the same as the average workload measured in the past few intervals. By averaging and "smoothing the ripples" in the workload, it tries to run the system at a constant optimal speed, thus reducing energy. This term takes the previous short-interval into account within the algorithm.
- The Derivative part of the equation (the third term) predicts that if the workload changes in the last interval, it is likely to change again at the same rate. This is the real predictive part of the equation, because it *anticipates* the changes that might occur in the workload and lets the operating system make a better choice of the required speed setting. This term takes the rate of change into account within the algorithm.

- The User Program Utilization part of the equation (the fourth term in RT-nqPID equation) gives the algorithm some headroom to find the optimal value for the speed setting. By setting the speed a little bit more than what the system requires, the Utilization part gives the voltage scheduler less choices of speed to set. The Utilization part limits the speed setting to a speed level which is enough to complete all tasks within their deadlines. Additionally, instead of trying to set the speed to only the next highest quantized value, the schedule that tries to set the speed to two or more quantized values so it can cause more energy saving as described in the work of Chandrasena [21]. The technique is called voltage quantized dithering. By using the individual task characteristics, this term takes the future into account within the algorithm.

By itself, each term is not efficient: the proportional part does not adequately study past behaviors, and so it cannot optimize power requirements. The integral part minimizes energy consumption at the cost of performance by not reacting fast enough. The derivative part cannot predict the actual workload, only changes in the workload, so it is unsuitable for steady-state operations. Finally, the utilization part is efficient to find an optimal speed to a set of tasks, but it does not consider other possible changes in workload, such as the OS overhead. However, the consensus of all of these can provide a very good

estimate of what the next value of the workload is likely to be. We take all three effects into account by taking a weighted sum of them.

CHAPTER 4

EXPERIMENT SETUP

4.1 Experimental Set-up

For the experiment, Motorola's 32-bit M-CORE architecture will be used as the model architecture [49]. This architecture is chosen because it was designed for good embedded-system performance and very low power operation. For the operating system, we use μC/OS II [50], a multitasking preemptive real time operating system. It is chosen to represent sophisticated preemptive multitasking RTOSs with footprints small enough for microcontroller systems to utilize. The operating system is extended to support monitoring system utilization. We also made modifications to the OS kernel that enable us to choose among the system with no voltage scaling, an nqPID algorithm, an RT-nqPID algorithm, the AVGN algorithm [20], and the HPASTS algorithm [46]. The AVGN algorithm is chosen to represent an algorithm using only the past utilization of the system. On the other hand, the HPASTS algorithm is represented the algorithm using the acceptance-test utilization.

We used several benchmarks from the MediaBench suite [48]. To assess the effects of an increasing workload, readings were taken with different numbers of tasks running simultaneously, and with different periods.

4.2 The M-CORE architecture and simulator

The M-CORE microRISC architecture was developed to achieve the requirement of the lowest milliwatts per MHz [51]. Its instruction set is designed to be an efficient target for high-level language compilers in terms of code density as well as execution cycle count. Variation of the integer data types (8, 16, and 32-bits) are supported for application migration from existing 8 and 16 bit microcontrollers. A standard set of arithmetic and logical instructions are provided, as well as instruction support for bit operations, byte extraction, data movement, and control flow modification. The processor provides hardware support for certain operations which are not commonly available in low-cost microcontrollers, including single-cycle logical shift, arithmetic shift, and rotate operations, a single cycle find-first-one instruction, a hardware loop instruction, and instructions to speed up memory copy and initialization operation. To maximize real-time control loop performance, the processor also has hardware support for multiplication and division.

Considering high code density, the M-CORE architecture adopts a compact 16-bit fixed length instruction format, and a 32-bit Load/Store RISC architecture. To minimize the power dissipation, the core is support for low-power instructions, DOZE, WAIT, and STOP mode. The system disables all

unnecessary peripherals in DOZE mode, while all activities are disabled in STOP mode. WAIT mode disables only the CPU, leaving peripheral functions active for short-term idle conditions. The architecture is operated under 2.0 volt and up to 40 MHz. The first implementation is targeted for cellular applications.

To minimize power consumption in the clock system, the core also adopts 1) clock gating, 2) delay clock, and 3) clock tree optimization. The speed, area, and power options are optimally selected to meet the timing constraints from a set of solutions from a hill-climbing search. Thorough research was conducted to determine every single process in datapath designs.

The simulator of the M-CORE architecture is used in our experiments. The simulator is a part of SimBed [49], a high-level-language model of an embedded hardware system that runs unmodified real-time operating systems. All devices, interrupts, and interrupt handlers used by the operating systems and applications are accurately simulated. The model has been verified as cycle-accurate to within 100 cycles per million compared to the actual hardware. The processor simulator is modeled to measure energy consumption within a difference of 10-15% different from the real measurements. The M-CORE simulator allows 36 voltage scaling levels, corresponding to clock frequencies from 2 to 20MHz. To keep track of the time with the real world, the simulator also provides an external timer. The

external timer is not changed its speed due to the voltage-frequency changes. So, the operating system can maintain constant voltage-scaling interval.

Finally, we assume a practical microprocessor [40], which cannot process data while its core voltage or operating frequency is changing. It takes a finite amount of time, and a finite amount of energy, to effect this change. Changing voltage levels and clock frequencies lower performance and has a cost in terms of energy that must be made up by the energy it saves. The amount of clock transitions taken in order to change from one voltage step to another linearly relates to the difference between those voltage steps. And, the time taken for the change is the ratio of the clock transitions to the current frequency. In the worst case, the processor requires 250 microseconds to change from the lowest clock frequency to the highest. The energy consumed during the transition is modeled as if the processor is executing the most energy-intensive instructions.

4.3 μC/OS-II

μC/OS-II is a preemptive multitasking RTOS that is in the public domain [50]. It is a highly portable, ROMable, scalable kernel, and has been ported to more than 40 different processor architectures ranging from 8- to 64-bit CPUs. Execution times of all kernel functions and services are deterministic, nevertheless, it is written in ANSI C for maximum portability. Despite its small size, it provides such services as mailboxes, queues, semaphores, time-related

functions, etc. It is chosen to represent sophisticated preemptive multitasking RTOSs with footprints small enough for microcontroller systems.

The scheduler of the μC/OS-II has 2-level table-lookup mechanism. There is a 8-bit first-level bit-vector that indicates which of these priority groups contains a task that is ready-to run. First, the scheduler looks for a ready task group from this bit-vector. One bit in this first-level bit-vector represents one task group. Therefore, the RTOS supports for 8 task groups, and each task group can have up to 8 tasks. If there are two or more group tasks ready, the scheduler will select the highest priority group. Then, the scheduler looks up in the 256-entry second-level table to determine which task in the selected group is ready. The 256-entry table is used in both determining the ready group and the ready task in a group. The priority of a task is determined by putting together the task number for the three least significant bits and task group number for the rest. Finally, the resulted priority is passed to the system's context switch module to be executed. Despite maintaining large look-up 256-entry table, this implementation has many advantages of executing in a deterministic amount of time since it does not directly depend on the number of tasks.

4.4 The MediaBench Suite

The MediaBench Suite [48] is a benchmark suite that has been designed to fill the gap between the compiler community for instruction level parallelism and

the embedded applications developers. The goals of the development are to accurately represent the workload of multimedia and communications applications, which are widely used in embedded systems. This kind of applications is well suited with the need of instruction level parallelism techniques.

MediaBench is composed of complete applications coded in high-level language. All applications are publicly available, making the suite available to a wider user community. The applications are ranging from image processing, communications and DSP applications. The application benchmarks used in our experiments include:

- **ADPCM** : Adaptive differential pulse code modulation is one of the simplest and oldest forms of audio coding. It is a speech compression and decompression algorithm. It is commonly implemented by sampling 16-bit linear PCM and converting them to 4-bit samples. The output is a compression rate of 4:1.
- **GSM**: GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate) into 260 bits. However, for compatibility with typical UNIX applications, the MediaBench implementation compresses frames of 160 16-bit linear samples into 33-byte frames (1650 Bytes/s). The quality of the algorithm is good enough for reliable speaker recognition; even music often survives

transcoding in recognizable form (given the bandwidth limitations of 8 kHz sampling rate).

For our experiments, we choose the period of 80 milliseconds and 160 milliseconds for ADPCM Encode, and 72 milliseconds and 120 milliseconds for ADPCM Decode. For GSM applications, we choose 80 and 160 milliseconds for the periods of the decode, and 100 and 200 milliseconds for the periods of the encode. The shorter periods are chosen to exhibit the near-overload or completely overload conditions. The longer periods are chosen to exhibit the general behavior of the system under a variety of numbers of tasks.

4.5 The AVGN Algorithm

The AVGN algorithm is a subset and simplification of Govil's AGED_AVERAGES algorithm and reminiscent of Weiser's PAST algorithm. Under AVGN, an exponential moving average with decay N of the previous intervals is used. That is, at each interval, it computes a "weighted utilization" at time t:

$$W[t] = \frac{N \times W[t-1] + U[t-1]}{N+1}$$

which is a function of the utilization of the previous interval U[t]. The AGED_AVERAGES algorithm allows any geometrically decaying factor, not just N/N+1.

Grunwald indicated that AVGN cannot settle on a clock speed that maximizes CPU utilization [22]. The set of parameters chosen could result in an optimal performance for a single application, but these parameters may not work for other applications. However, they found that the AVGN policy resulted in both the most responsive system behavior and the most significant energy reduction of all the policies they examined.

Care was taken to implement the AVGN algorithms faithfully. All parameters other than the algorithms themselves were kept the same to ensure a fair algorithm-to-algorithm comparison. The AVGN algorithm was set to change speed whenever the system utilization drifted out of its optimum range of 50-70%, and a value of 3 will be used throughout the experiments.

4.6 The HPASTS algorithm

The HPASTS algorithm was introduced by Hong et al [46]. The algorithm considers the task characteristics to determine the appropriate setting speed. It adopts the concept of total utilization as in the acceptance test using widely in RTOS. The heuristic accounts for both periodic tasks and sporadic tasks in the system. Since we consider only periodic tasks in our experiment, the algorithm calculates the speed as equal to the sum of the utilization (the ratio of a task's execution to its period) of all periodic tasks in the system.

We will show in the next chapter that even though the heuristic gives near-perfect results in the case containing small number of tasks. However, when the number of tasks is large, the heuristic set the processor speed too low, then causes the lowest-priority task to miss its deadline.

4.7 The Experiment-Setup Details for nqPID and RT-nqPID

The nqPID coefficients were chosen to reflect a middle-of-the road configuration that would not be fine-tuned to any particular benchmark. We select the value of each parameter as m=10, Kp=0.4, Ki=0.4, and Kd=0.2. For the RT-nqPID algorithm, we select the same parameter values as in the nqPID with Ku=0.36.

For the control rate, the voltage scheduler computes the desired speed every 5 milliseconds. Even though a small voltage interval means fast response, a small interval also means more frequent computation and higher processor-time demand. Since the periods of the tasks are in order of a hundred milliseconds, the voltage scheduling rate of every 5 milliseconds, which is around a tenth to a twentieth of the periods, should be reasonable as described by Nyquist sampling theorem. Additionally, the voltage interval should be large enough compared with the voltage-scaling transition time. For the M-CORE simulator, the worst-case voltage-scaling transition time is 250 microseconds, which is accounted for only 5% of the 5-millisecond interval. Since the 5-millisecond interval is large

enough for the voltage transition overhead and it is small enough for the task periods, 5 milliseconds should be sufficient for our voltage-scaling interval to be used within the system.

CHAPTER 5

RESULTS

5.1 Energy and Performance

Our experiments use benchmark programs from the MediaBench suite of embedded-systems applications [48] that we have ported to the μC/OS-II embedded operating system. To vary the load on the system, two different periods were used for each benchmark, and up to 8 tasks were run simultaneously. Each task refers to a producer-consumer pair of processes. To measure the efficiency of the voltage-scaling heuristics, we measure total processor energy consumed by the system over a set number of user-level instructions and the variability in a task's execution time in terms of jitters and miss deadline rates. Energy is reported as a percentage of the energy of the non-DVS-enabled system. A jitter is defined as the variation in the period of all tasks due to the variation of the execution time. A jitter is reported as a percentage of the task's desired period. Finally, the task miss-deadline rate is reported as a percentage of the expected number of tasks completed in the simulation interval. Note, many real-time control systems require accurate timely behaviors upon a task completion. An acceptable jitter and a miss-deadline rate should be around a

percent or two [18], while more than a few percent of those can make the system unusable.

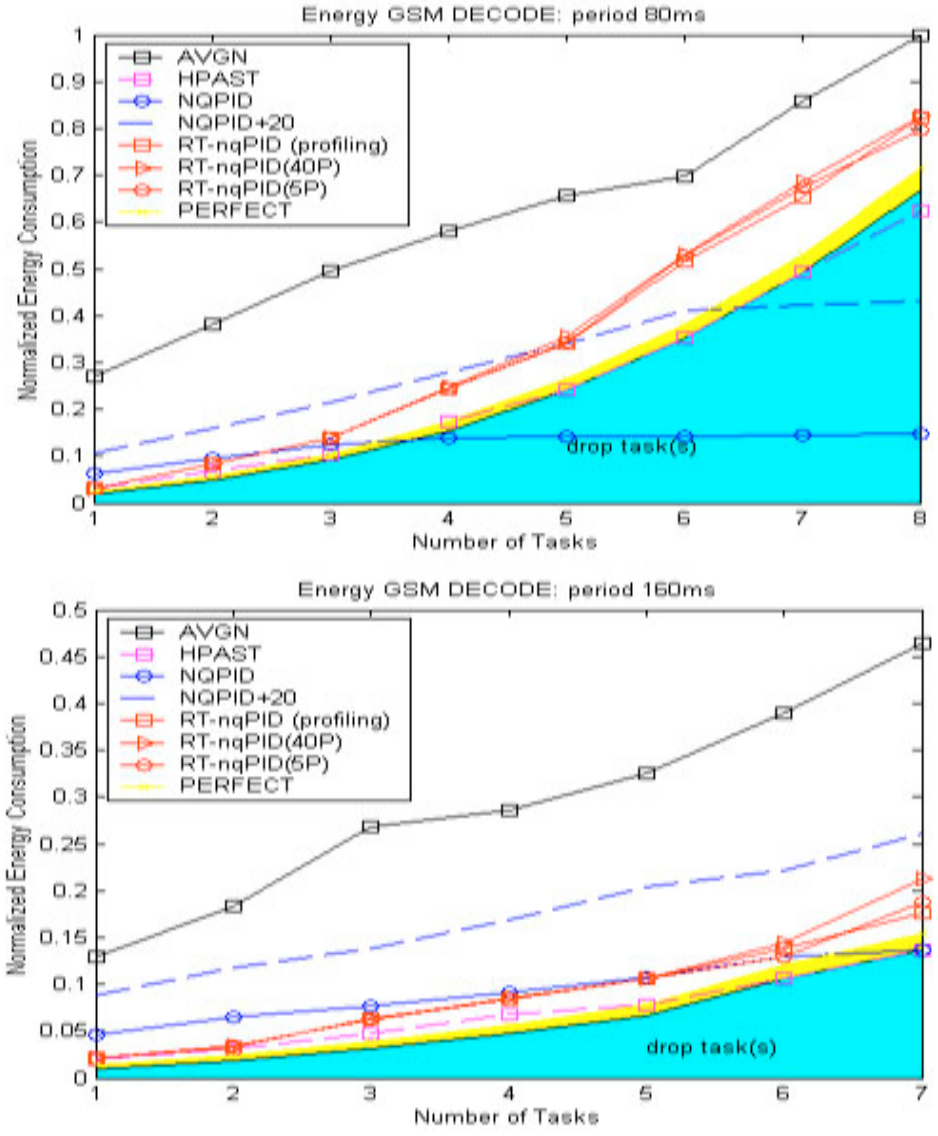

Figure 4. Energy Comsumption for GSM_DECODE. The upper figure shows the normalized energy consumption for 80ms-period tasks and the lower figure shows the normalized energy consumption for 160ms-period tasks. The y-axis is the normalized energy consumption with the non-DVS-enabled system. The x-axis is the number of tasks in running simultaneously.

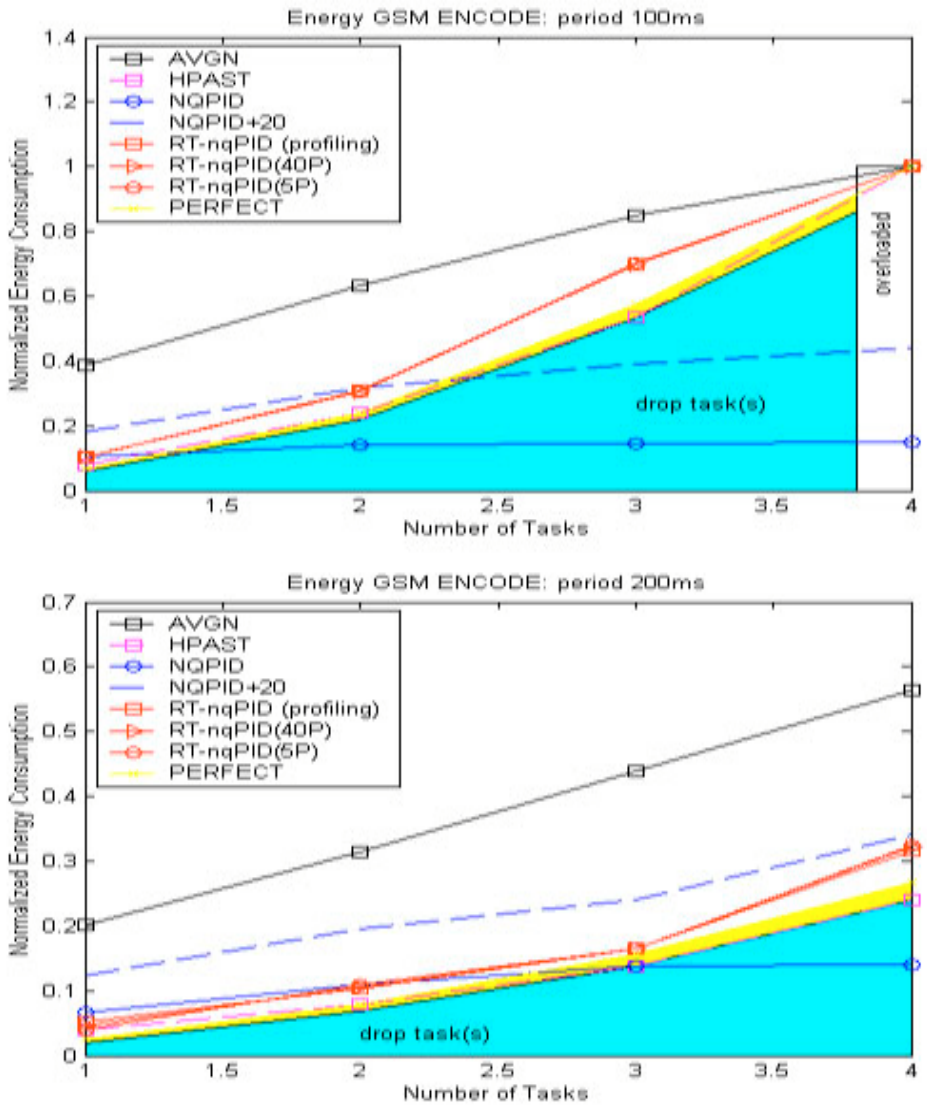

Figure 5. Energy Comsumption for GSM_ENCODE. The upper figure shows the normalized energy consumption for 100ms-period tasks and the lower figure shows the normalized energy consumption for 200ms-period tasks. The y-axis is the normalized energy consumption with the non-DVS-enabled system. The x-axis is the number of tasks in running simultaneously.

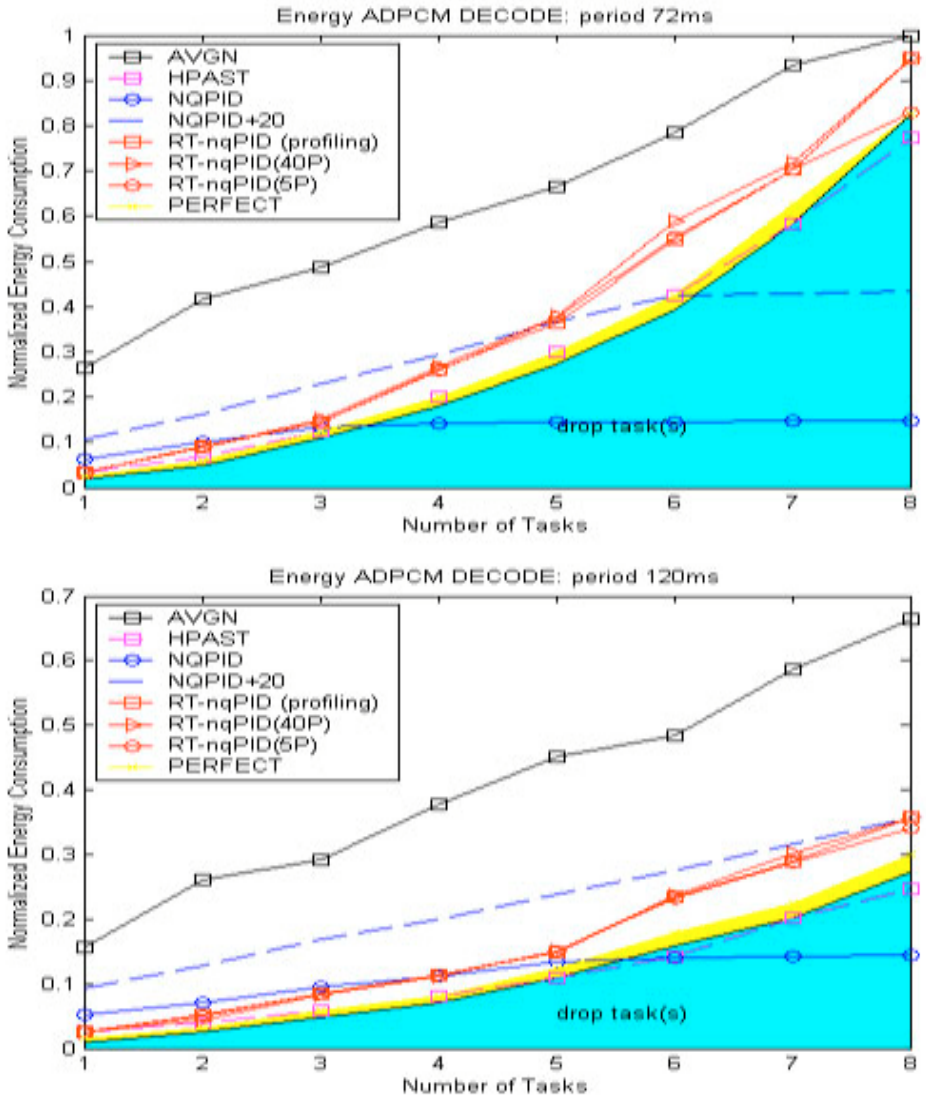

Figure 6. Energy Comsumption for ADPCM_DECODE. The upper figure shows the normalized energy consumption for 72ms-period tasks and the lower figure shows the normalized energy consumption for 120ms-period tasks. The y-axis is the normalized energy consumption with the non-DVS-enabled system. The x-axis is the number of tasks in running simultaneously.

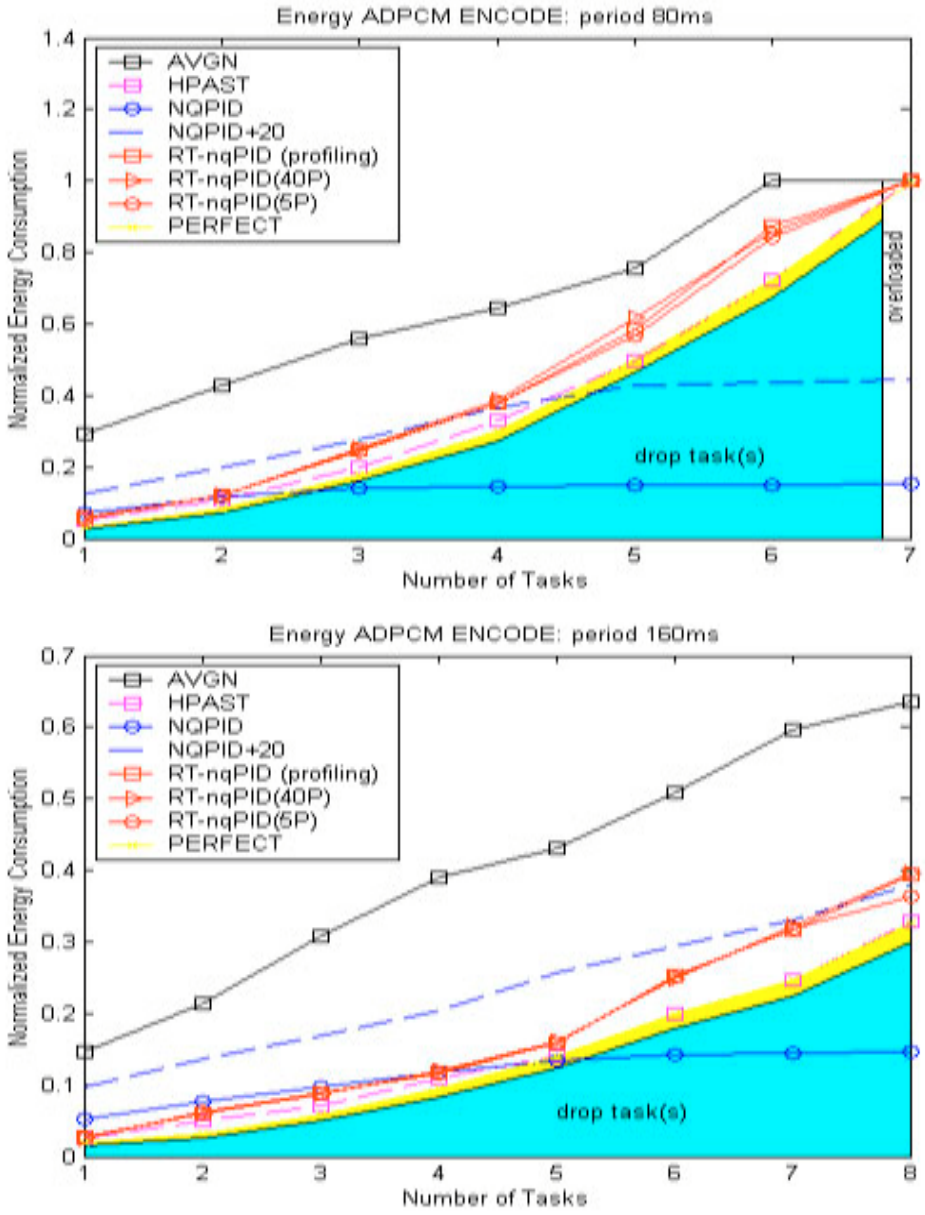

Figure 7. Energy Comsumption for ADPCM_ENCODE. The upper figure shows the normalized energy consumption for 80ms-period tasks and the lower figure shows the normalized energy consumption for 160ms-period tasks. The y-axis is the normalized energy consumption with the non-DVS-enabled system. The x-axis is the number of tasks in running simultaneously.

Figure 4, Figure 5, Figure 6, and Figure 7 compare the energy consumption of the RT-nqPID (labeled with profiling), AVGN, and HPASTS algorithms. The x-axis is the number of tasks running simultaneously, and the y-axis is the energy consumption normalized to non-DVS-enabled system. The PERFECT line is the energy consumption of the speed settings that is set to a constant speed that has an acceptable jitter and overall miss-deadline rate (below 5% for both of them). Since speed dithering may produce better results than the PERFECT line, we conducted another simulations using a constant speed of one step less than the PERFECT speed. Therefore, any algorithm consuming the energy below the one-step-lower-than-PERFECT line would have an unacceptable timely behavior, i.e. unacceptable jitters and/or miss-deadline rate. That unacceptable area is labeled as the 'drop task(s)' area. Additionally, the optimal algorithm should have its energy consumption somewhere below the PERFECT line but above the 'drop task(s)' area.

Figure 4 shows the energy consumption of all algorithms for the GSM decode applications, and Figure 5 shows the energy consumption for the GSM encode applications. The energy consumption graphs for the ADPCM decoder applications are shown in Figure 6, and the graphs for the ADPCM encoders are shown in Figure 7. In each of the figures, the upper graph represents different task periods than the lower graph. For example, in Figure 4, the upper graph

represents task periods of 80 milliseconds, and the lower graph represents task periods of 160 milliseconds. Each graph shows results for up to 8 simultaneous executing tasks . The area labeled 'overloaded' represents system-overloaded workload configuration.

The RT-nqPID algorithm reveals impressive behaviors across all benchmarks and load configurations: the energy consumption lines of the RT-nqPID algorithm are lying above the PERFECT line with only 4.68% gap on average. It gives the near-perfect results in case of small workloads. Compared with AVGN, the systems with RT-nqPID algorithm consume only 46.3% of the energy that is consumed by those with AVGN on average. On a light workload, the RT-nqPID consumes as low as 10% of the energy consumed by AVGN. Additionally, the algorithm preserves real-time behaviors of the applications by neither allowing any task to miss the deadline nor allow their jitters to go over 3%. This behavior is shown in the graphs since the energy consumption of the RT-nqPID never drops into the 'drop task(s)' area.

On the other hand, the HPASTS algorithm also produces near-perfect results in the case of a light workload. Nevertheless, as the number of tasks grows, it causes the system to drop tasks. Table 1 shows the percentage of missed deadline jobs containing low priority tasks, and table 2 shows the percentage of jitters across all benchmarks. The shaded areas represent the percentages that

shows unacceptable timely behaviors since the HPASTS algorithm does not take the operating systems overhead into account. While increasing the number of tasks causes the operating system overhead to grow, the algorithm still selects the same constant speed. As a result, the algorithm cannot preserve real-time task behaviors.

#task	GSM decode		GSM encode		ADPCM decode		ADPCM encode	
	80ms	160ms	100ms	200ms	72ms	120ms	80ms	160ms
1	0.6	1	0.4	0.9	0.6	0.7	0.4	0.8
2	0.7	2	4.1	2.5	0.5	1	0.02	1.6
3	0.6	1.2	11.7	14.6	1	1.4	1	0.4
4	3	1.2	O/V	16	0.9	32.7	0.8	1.2
5	32.4	25.3			42.9	56.9	0.6	1.8
6	35.2	33.3			51	84.2	0.5	0.8
7	39	43.3			53.7	68	O/V	2.4
8	66.9	51.7			57.2	94.6	O/V	2.3

Table 1. The Percentage of Miss Deadlines for HPASTS Algorithm. The value in the table is the percentage of the lowest-priority task missing its deadline. O/V means an overloaded workload configuration. Unacceptable values, exceeding 5%, are shaded. A black cell represents a configuration with no experiment conducted.

#task	GSM decode		GSM encode		ADPCM decode		ADPCM encode	
	80ms	160ms	100ms	200ms	72ms	120ms	80ms	160ms
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	1.57	0.00	0.00	0.00	0.00	0.01
3	0.00	0.01	3.50	4.07	0.01	0.00	0.01	0.01
4	0.43	0.00	O/V	2.98	0.00	8.06	0.00	0.00
5	6.46	4.60			3.81	11.64	0.00	0.00
6	6.02	5.51			4.67	13.87	0.07	0.00
7	5.50	5.72			5.18	10.14	O/V	3.54
8	8.77	6.56			5.84	7.40	O/V	0.00

Table 2. The Percentage of Jitters for HPASTS Algorithm. The value in the table is the percentage of the jitters compared with the period of the tasks. O/V means overloaded workload configuration. Unacceptable values, exceeding 5%, are shaded. A black cell represents a configuration with no experiment conducted.

In addition to the original RT-nqPID algorithm, we conducted more experiments to find the worst-case execution time for each task. Instead of using preset worst-case execution time from profiling information, we ran the benchmark applications with maximum speed for a number of task periods to find the worst-case execution time. This interval is called the learning phase. Then, the RT-nqPID uses that execution time to determine the speed in the algorithm as usual. We set the learning phase to 5 and 40 task periods. The results are shown in Figure 4, Figure 5, Figure 6, and Figure 7 with the line labeled with RT-nqPID 5P and 40P for the RT-nqPID algorithm with 5 task periods and 40 task

periods for the learning phase, respectively. The modified algorithms still give effective results: the energy consumption graphs of both cases are very close to the original RT-nqPID across all benchmarks. We can conclude that the learning phase can take only a little amount of time to make the algorithm as efficient as the original algorithm. Therefore, the RT-nqPID algorithm is a practical and an effective solution to the DVS problem in real-time embedded system.

5.2 The Comparison of nqPID and RT-nqPID algorithms

The PID controller has been widely used to solve problems in control systems for decades. The controller has an ability to track the input value by using the error between the measured output and the reference value. It is acceptable as an efficient method to solve most problems in any control system.

Even though only the nqPID function, using only the adaptation of the PID controller, can correctly track the system utilization, but it is not enough for real-time applications. The nqPID function exhibits unacceptable real-time behaviors by dropping tasks when the workload is increasing. As a result, we need more than just the nqPID function, namely the RT-nqPID algorithm.

This section shows how the nqPID function behaves when the number of tasks increases. Then, we display the behaviors when the headroom constant is increased. The results show that, even increasing the headroom constant, the

nqPID cannot preserve the real-time behaviors. Finally, we show the proposed RT-nqPID behavior and how it maintains the real-time behavior.

First, we implement the adaptation of the PID control algorithm, the nqPID algorithm, as described in chapter 3 and chapter 4. We also added the algorithm's speed settings with 2 different constants to generate headroom. First, we add a constant of 10% of the maximum speed (about 2 MHz) to the calculated results. Figure 8 shows the speed settings of the nqPID algorithm with 10% headroom constant. In this case, the figure shows the speed setting of 1 to 8 GSM decode tasks running simultaneously with 160-millisecond task period. None of the experimental load configurations are overloaded. The x-axis is the time in milliseconds, and the y-axis is the processor speed in MHz. The speed setting is oscillating between 4 MHz and 8.5 MHz in every case. However, in the case of 7 and 8 tasks running simultaneously, the system demonstrates the saturated state as described in chapter 3. The speeds settle at 8.5 MHz with 6.95% jitters and 52% miss deadline rate of the lowest-priority task in the 7-task case, and 8.61% jitters and the system totally discards the lowest-priority task in the 8-task case.

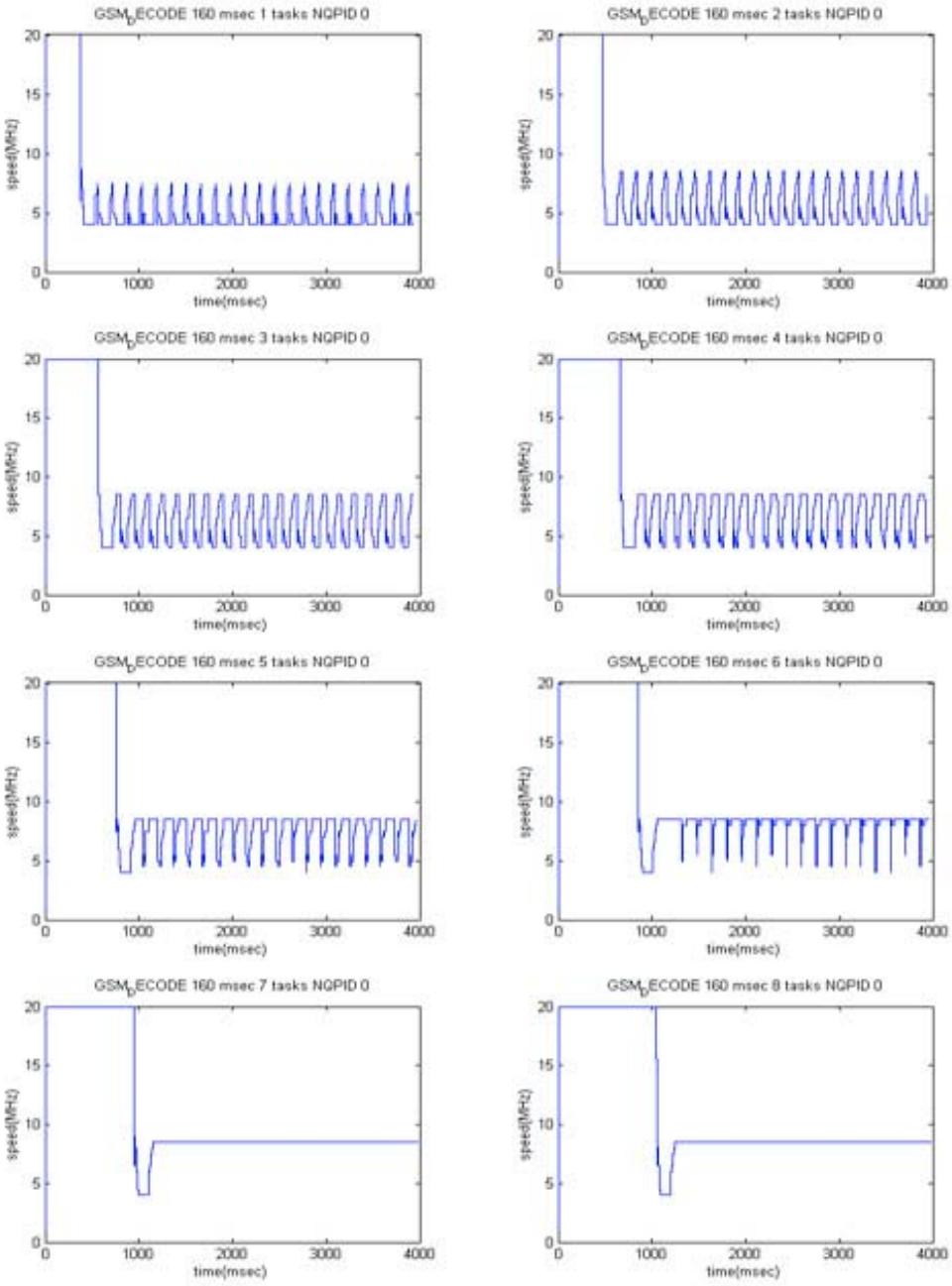

Figure 8. Speed setting for nqPID algorithm for GSM DECODE 160ms-period. The x-axis is the time in milliseconds, and the y-axis is the processor speed in MHz. The speed is varied in the same range in all cases. In case of 7 and 8 tasks running simultaneously, even though the speed is not the highest, the saturated state occurs.

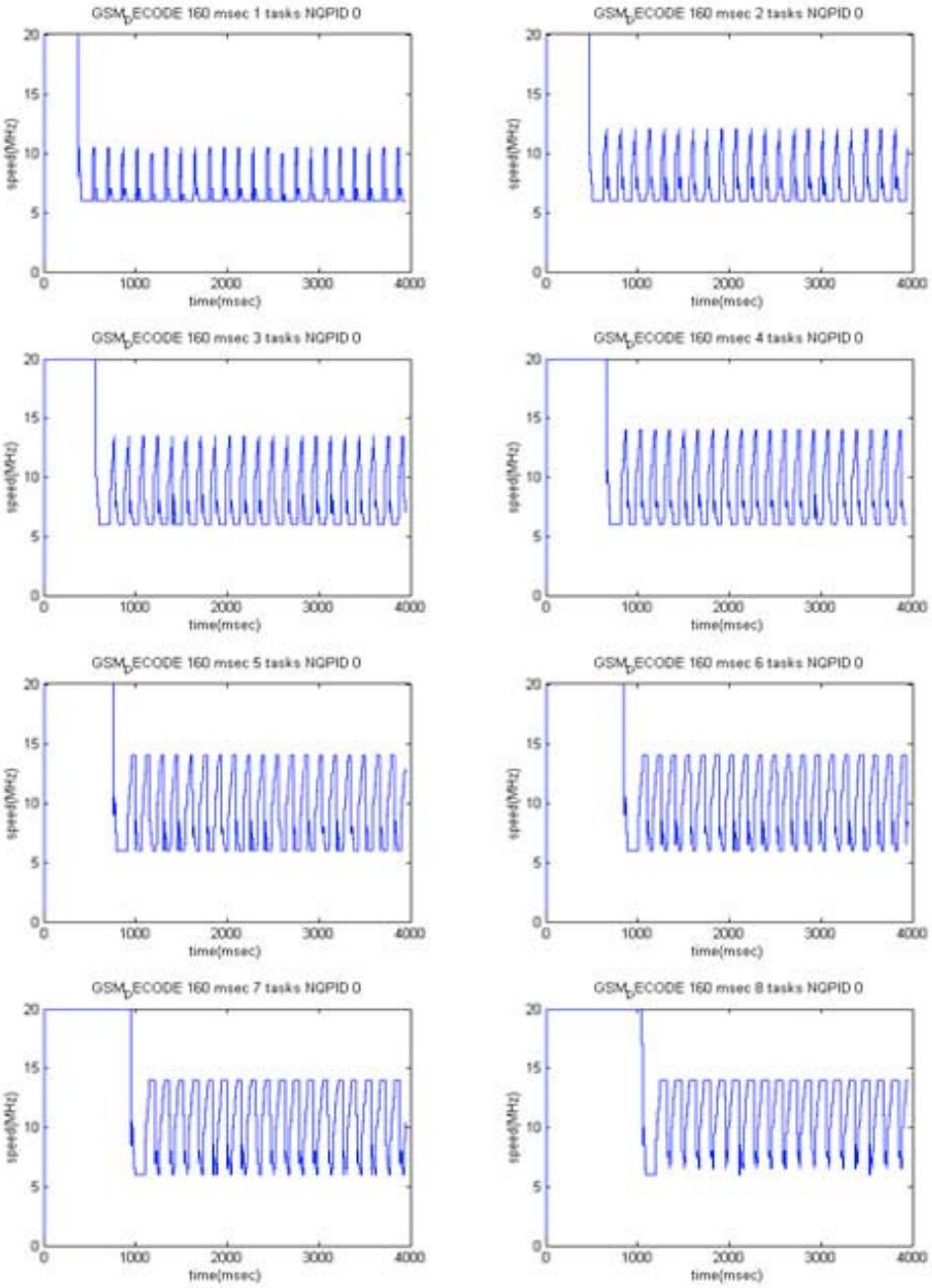

Figure 9. Speed setting for nqPID algorithm with 20% headroom for GSM DECODE 160ms-period. The x-axis is the time in milliseconds, and the y-axis is the processor speed in MHz. The speed is varied in the same range in all cases. All miss-deadline rates are below 5 per cent.

Next, we test with an added headroom constant of 20% to the calculated results. Figure 9 shows the speed setting for the later headroom constant. In the case of 20% headroom, the speed setting is oscillating between 6MHz and 14MHz, wider range, without any occurrences of saturated state in this particular benchmark. However, the energy consumption in the 20% headroom case is much more than the 10% case in all benchmarks as shown in Figure 4, Figure 5, Figure 6, and Figure 7. In the case of short task period in all benchmark applications, as we can see here that the energy graphs lie in the drop-tasks area, even adding 20% headroom cannot guarantee real-time constraints. As a result, increasing the headroom constant only postpone the unacceptable real-time behavior to some configuration containin greater number of tasks.

The disadvantage of the nqPID algorithm is it cannot detect system-overloaded condition. . The algorithm sets the speed oscillating between smaller ranges, but it will show unacceptable real-time behavior even when the headroom constant is increased. The reason is the maximum speed allowed in the range does not scale with the system utilization. The allowed maximum speed is always 8.5 MHz in the 10% headroom case, and always 14 MHz in the 20% headroom case. Therefore, we need a mechanism to increase the allowed maximum speed when the number of tasks grows.

As a result, we set the headroom constant related to the total utilization of the system as described in chapter 3. The experiment results of the new algorithm, RT-nqPID, in the case of running 1 to 8 GSM decode tasks simultaneously with 160-millisecond task period are shown in Figure 10. Obviously, with the constant headroom adjusted by the application parameters, the speed settings are smoother than those in the case of 10% and 20% headroom are. The maximum speed allowed is scaled with the number of tasks: while the number of tasks is increasing, the maximum speed also increases. The energy consumption is above the perfect case with only 5% different on average (see Figure 4).

Therefore, the RT-nqPID algorithm, which is the adaptation of the nqPID, is able to maintain real-time behaviors of all benchmark applications. The algorithm inherits the advantageous properties from the nqPID as it preserves smoothness, reacts to changes quickly, and learns from the past measured variables. Additionally, since it takes the future information into consideration via system utilization computations, it can set the system speed to meet the real-time constraints.

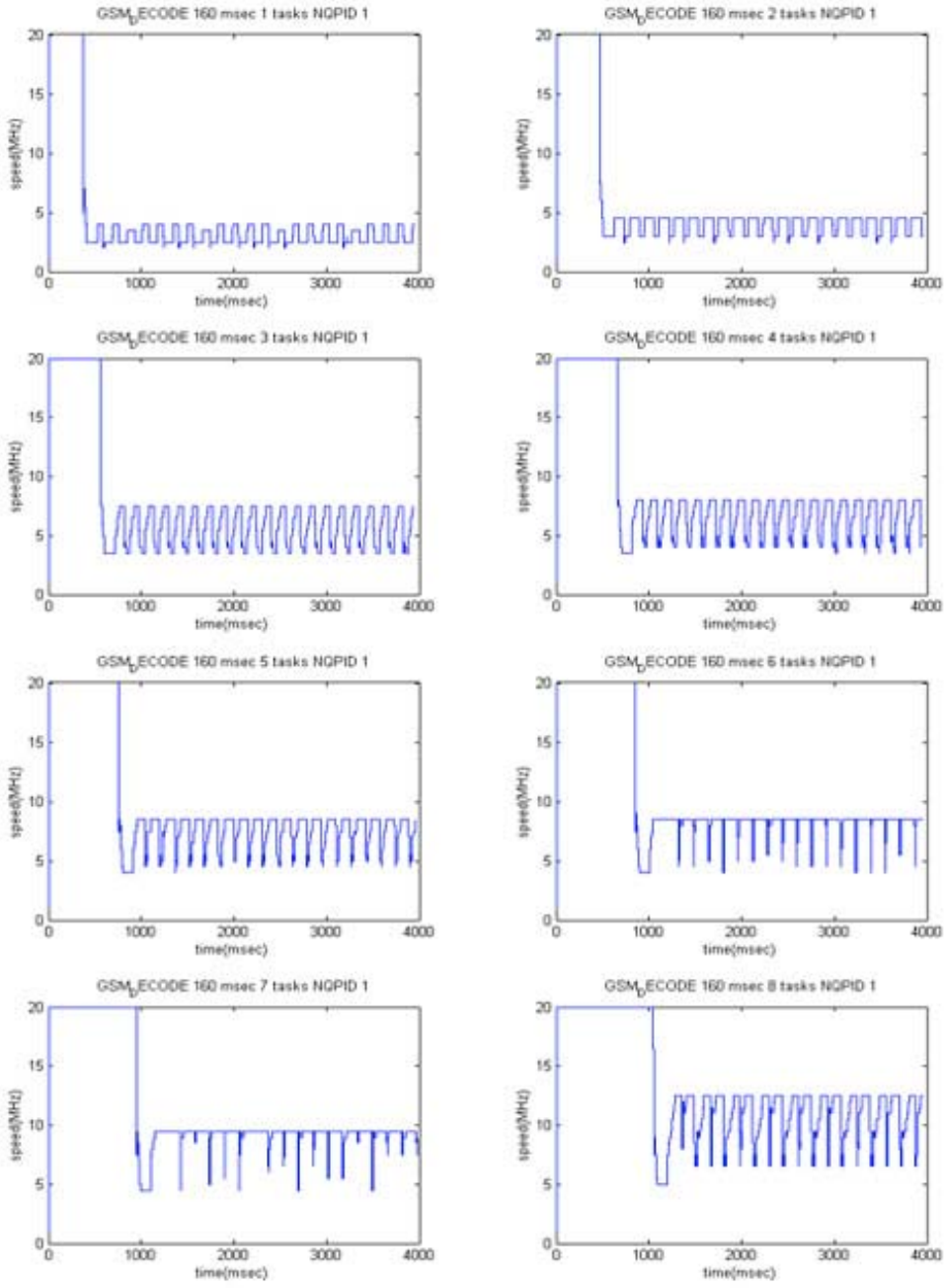

Figure 10. Speed setting for RT-nqPID algorithm for GSM DECODE 160ms-period. The x-axis is the time in milliseconds, and the y-axis is the processor speed in MHz. The speed setting range is varied in each case. All miss-deadline rates are below 5 per cent.

5.3 The Run-Time Trace

To compare the behaviors of the three algorithms, RT-nqPID, AVGN, and HPASTS, we took snapshots of the system during execution. Figure 11, Figure 12, and Figure 13 show how each algorithm responds to system load over the interval of one second. The graphs show 1 to 8 ADPCM decode tasks running simultaneously with a 72 -milliseconds task period. As described in the previous section, the x-axis is the time in milliseconds, and the y-axis is the speed in MHz. Figure 11 is the speed setting behavior of the RT-nqPID algorithm. Figure 12 and Figure 13 show the speed setting behaviors of AVGN and HPASTS algorithms, respectively. Since the voltage level is scaled proportionally with the processor speed, the speed setting graphs also indicate the voltage level at which the processor is running.

We derived these conclusions by inspecting the speed setting graphs:

- The AVGN algorithm cannot converge to an optimal speed and continues oscillating between the lowest speed and the highest speed. This behavior of the algorithm is expected, and it was analyzed by Grunwald [22].
- The HPASTS algorithm sets the speed to a constant, thus the speed setting is smooth for the entire interval. However, the algorithm reveals unacceptable real-time behaviors as shown in table 1 and in table 2.

- The RT-nqPID algorithm tries to find the optimal speed in a range. It demonstrates oscillation between a narrow range, and the oscillating frequency is much smaller than the frequency in AVGN. The proposed algorithm settles down fast as soon as the workload stabilizes as it correctly tracks the system's workload. The rate of change in the derivative term helps the algorithm keep up with the change of the workload in both positive and negative directions.
- In the case of running 8 tasks simultaneously, the workload is nearly overloaded. The AVGN algorithm sets the speed to the maximum, while the RT-nqPID still keep s trying to reduce the speed. As a result, the RT-nqPID can reduce more energy consumption.

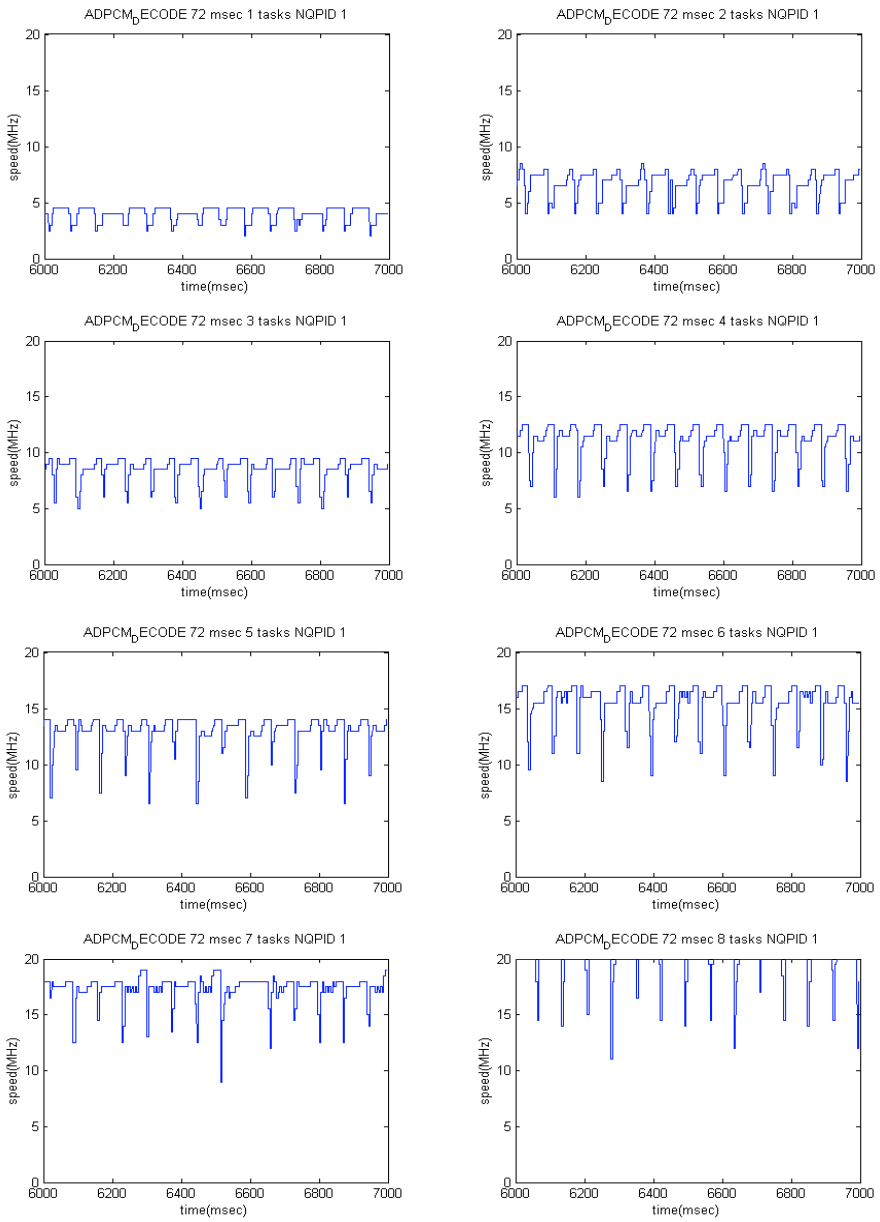

Figure 11. Speed setting for RT-nqPID algorithm for ADPCM DECODE 72ms-period. Even though the speed setting is oscillating, the range is much smaller than the AVGN algorithm, and the frequency of oscillating is much less. Additionally, the miss deadline rate is always below 5% in every case.

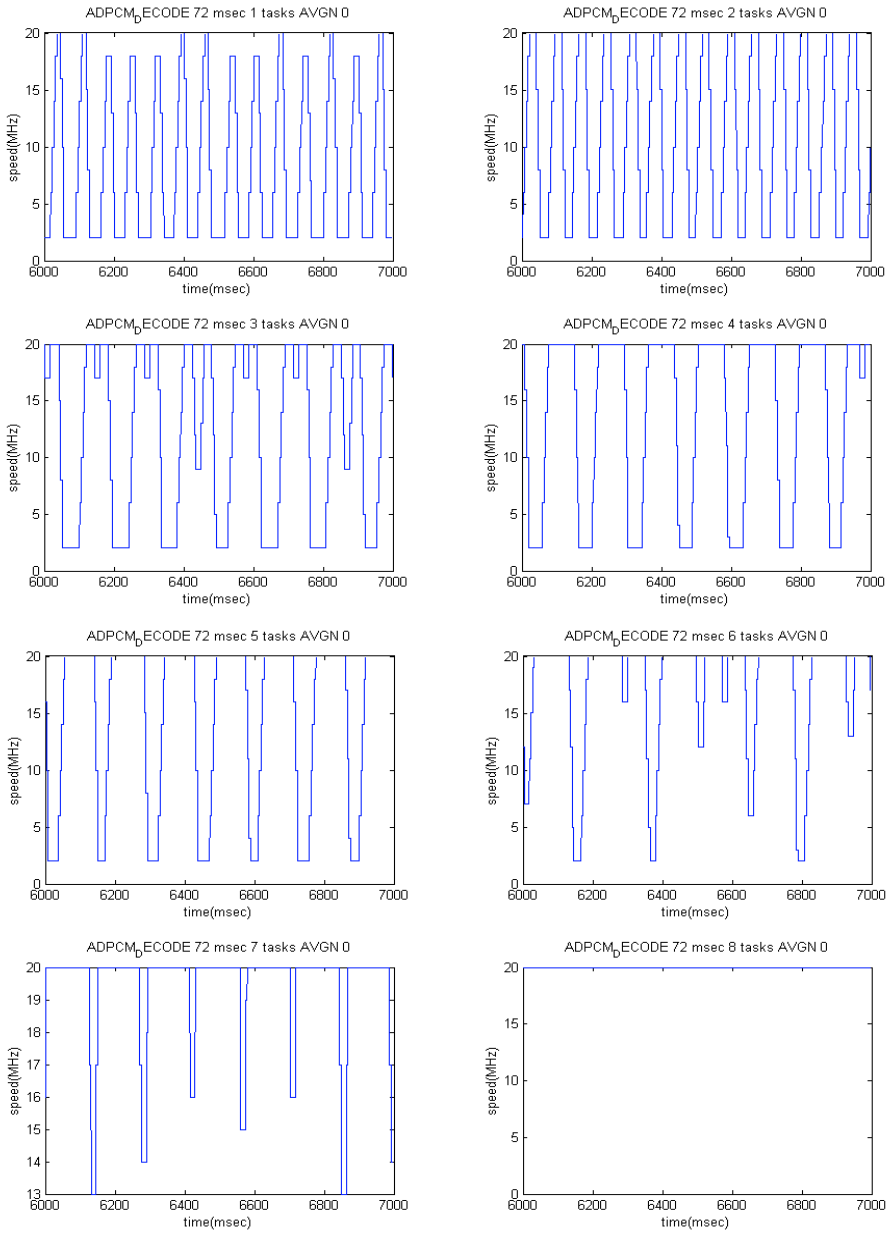

Figure 12. Speed setting for AVGN algorithm for ADPCM DECODE 72ms-period. The speed is oscillating between the minimum speed and the maximum speed. It never settles in an optimum speed as analyzed by Grunwald.

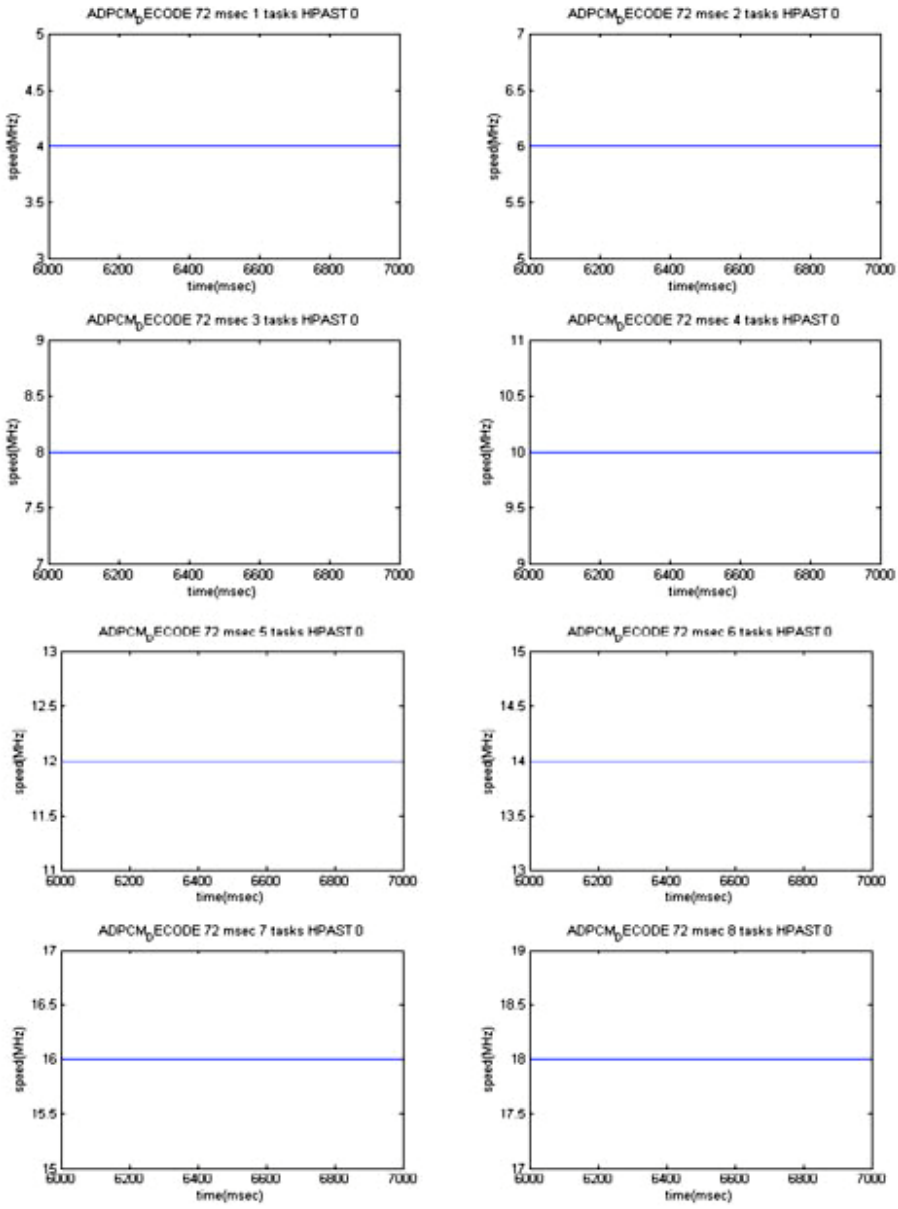

Figure 13. Speed setting for HPASTS algorithm for ADPCM DECODE 72ms-period. The speed settles at an optimum speed. However, the lowest-priority task misses its deadline when the number of task are from 5 to 8 (42.9% , 51.0%, 53.7%, and 57.2%, respectively).

5.4 Sensitivity Analysis

The different choices of coefficients in the RT-nqPID algorithm can make one to misinterpret the efficiency of the algorithms. For each coefficient in the nqPID function, we conducted an experiment by running the system with the sets of K_P, K_I, and K_D in table 3 and table 4. Note, the sum of K_P, K_I and K_D are equal to 100% of the predicted workload to represent a planar cut through the design space and to simplify design exploration. The K_D is not shown for simplification. One can calculate K_D by simply subtracting 100 with K_P and K_I. The duty of the nqPID function is to track and predict the system workload; therefore, the output from the function should be bounded by the maximum workload, which is 100%.

Given that the miss-deadline rates and the jitters must be lower than 5%, we choose the value K_U as small as possible so that the system only maintains the real-time behaviors. The user program utilization part will add up with the nqPID function to push the workload in the future back to the current interval. Since the excessive workload procrastinated to the future causes the system to miss deadlines, the user program utilization part makes sure that no excessive workloads are carried to the next cycle. However, the coefficient K_U has to be minimal to prevent the system to be too conservative and to cause idle clock cycles in the system.

The table 3(a)-(d) shows the value of the chosen K_U for a particular configuration, and the table 4(a)-(d) show the average energy consumption for each configuration. The numbers shown are reported as the percentage of the energy consumed by the non-DVS system. As the tables show, the energy consumption is relatively in the same range, i.e. only 15% difference from the maximum value and the minimum value. No configurations significantly consume more energy than others do when running with the same application. We conclude that, even though, the choice of coefficients can vary the performance for a particular benchmark, we would like to present the algorithm fairly with respect to others. Therefore, we choose to implement our algorithm with the middle-of-the-road configuration: K_P=40%, K_I=40%, K_D=20%, and K_U=36%.

Kp/Ki	0	20	40	60	80	100
0	100	100	76	76	38	17
20	100	90	70	40	23	
40	65	55	36	20		
60	63	45	25			
80	35	23				
100	22					

(a) The value of Ku for GSM Decode

Kp/Ki	0	20	40	60	80	100
0	100	100	100	52	37	16
20	100	100	55	35	14	
40	100	100	33	17		
60	99	30	12			
80	72	12				
100	38					

(b) The value of Ku for GSM Encode

Kp/Ki	0	20	40	60	80	100
0	100	100	76	63	37	22
20	80	80	60	33	20	
40	65	47	33	15		
60	55	32	17			
80	32	14				
100	14					

(c) The value of Ku for ADPCM Decode

Kp/Ki	0	20	40	60	80	100
0	100	96	77	56	39	27
20	79	74	55	40	23	
40	63	60	33	23		
60	55	35	18			
80	35	29				
100	20					

(d) The value of Ku for ADPCM Encode

Table 3. The value of Ku for each benchmark. Each benchmark requires different Ku values for different sets of the coefficients. Note, that in the case of GSM encode application, the algorithm shows unacceptable real-time behavior when Ku is 100.

Kp/Ki	0	20	40	60	80	100
0	18.56	22.46	21.22	27.21	20.96	21.11
20	25.86	26.16	26.12	22.32	23.06	
40	21.63	22.04	21.54	23.16		
60	28.96	26.47	26.79			
80	28.29	28.17				
100	33.00					

(a) Normailized Average Energy Consumption for GSM Decode

Kp/Ki	0	20	40	60	80	100
0	29.03	32.66	39.28	28.68	29.80	30.73
20	36.75	39.88	31.01	29.31	27.23	
40	42.54	47.03	30.80	32.24		
60	48.21	30.66	32.03			
80	47.43	34.08				
100	45.82					

(b) Normailized Average Energy Consumption for GSM Encode

Kp/Ki	0	20	40	60	80	100
0	23.01	27.98	26.15	27.73	24.47	27.48
20	23.39	28.12	27.59	23.89	26.11	
40	26.51	22.25	24.23	24.71		
60	32.24	25.29	29.20			
80	32.76	28.58				
100	36.74					

(c) Normailized Average Energy Consumption for ADPCM Decode

Kp/Ki	0	20	40	60	80	100
0	31.54	34.08	33.93	33.15	33.62	37.96
20	31.25	33.54	33.03	34.80	36.06	
40	33.59	36.29	32.53	36.87		
60	39.17	35.66	36.89			
80	41.20	42.94				
100	45.06					

(d) Normailized Average Energy Consumption for ADPCM Encode

Table 4.The percentage of Normalized Average Energy Consumption for each benchmark. The energy is normalized to the energy consumed by the no-DVS system. The difference between the maximum value and the minimum value is 15%.

CHAPTER 6

CONCLUSION

6.1 Summary

Minimizing power consumption is a growing concern since embedded processors are widely used in portable and inaccessible systems, operated by battery. Known fact, the core processor consumes a large portion of energy in such systems. Dynamic voltage scaling (DVS) is accepted as the key technique to reduce energy dissipation by lowering the supply voltage and operating frequency in response to the concept of performance on demand. Lowering the operating frequency in conjunction with the supply voltage can reduce a significant amount of energy because the quadratic relations between energy and voltage supply are so intertwined.

This thesis proposes a new approach to Dynamic Voltage Scaling heuristics, the nqPID and its real-time extension RT-nqPID, on an embedded microcontroller. The algorithms are adaptations of the classical control method, PID controller. The RT-nqPID can be used as a mechanism to find a good representative among the present, the past, and the changing workload of the system. Also, it settles to an optimum operating point fast and preserves smoothness. In addition to producing a good representative value, the RT-nqPID also preserves the real-time constraints.

The controller is implemented in an execution-driven environment: a simulation model of Motorola's M-CORE microcontroller that is realistic enough to run the same unmodified operating system and application binaries that run on hardware platforms. The simulation model is instrumented to measure performance as well as energy consumption. The controller is integrated into the embedded operating system that executes a multitasking workload. As a comparison, we also implement the AVGN and the HPASTS heuristics.

The experiment results show that the RT-nqPID algorithm can significantly reduce the energy consumption. The proposed algorithm can reduce the energy consumption to only less than 5% difference on average from the perfect case. In some cases, the RT-nqPID can produce results that are less than 1% difference. On the other hand, the best of the class, AVGN algorithm reduces the energy consumption by up to 40% difference from the perfect case, while our proposed algorithms use less than half of that energy. From the real-time behavior perspective, the RT-nqPID algorithm neither causes any tasks jitters to exceed 5% nor causes any tasks to miss their deadline over 3%. While the acceptance-test-based HPASTS algorithm causes the lowest priority task to miss its deadline up to 90% of the expected number of tasks completed in the simulation interval. Therefore, the RT-nqPID scheme outperforms both AVGN

and HPASTS algorithms in both energy consumption and performance as measured by miss deadlines in the periodic task's execution time.

In conclusion, the RT-nqPID algorithm is practical and effective solution to minimize energy consumption, specifically DVS, problems in embedded systems. It is designed to be independent from any task scheduling algorithm. In all cases, the RT-nqPID shows impressive energy reduction performance and real-time constraints guarantee.

6.2 Future Works

For the future works, we will explore the possibility to extend the RT-nqPID algorithm to support applications that have wide variation in execution such as MPEG. Additionally, we will also apply it to support the systems with aperiodic and sporadic tasks. More energy saving schemes, such as dynamic power management (DPM), will be applied along with an algorithm to further reduce energy consumption. Furthermore, we will explore using other 'average' techniques, such as finding an expected value in a probability model and applying a certain signal-processing filter in order to find a good representative value for the system at any interval.

REFERENCES

1. Schlett, M., *Trends in Embedded-Microprocessor Design.* IEEE Computer, 1998. 31(8): p. 44-48.

2. *Most Important consideration in Selecting an Embedded OS*, in *Embedded Systems Developer survey*. 2003, Evans Data Corporation.

3. Y. Li, M. Potkonjak, and W. Wolf, *Real-time Operating Systems for Embedded Computing.* Predeedings of the 1997 IEEE Internatinal conference on Computer Design: VLSI in Computers and Processors, 1997: p. 388-392.

4. Barr, M., *Moving Targets*, in *Embedded systems Programming*. 2003.

5. T. Pering, T. Burd, and R. Brodersen, *Dynamic voltage scaling and the design of a low-power microprocessor system.* Power Driven Microarchitecture Workshop, attached to ISCA98, 1998.

6. M. Horowitz, T. Indermaur, and R. Gonzalez, *Low-Power Digital Design.* Symposium on Low Power Electronics, 1994. 1: p. 8-11.

7. K. Govil, E. Chan, and H. Wasserman, *Comparing algorithms for dynamic speed-setting of a low-power CPU.* Proceedings of The First ACM International Conference on Mobile Computing and Networking, 1995.

8. Woonseok Kim, Dongkun Shin, Han Saem Yun, Jihong Kim, and Sang Lyul Min, *Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems.* Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002), 2002.

9. T. Burd, R. Brodersen, *Design issues for Dynamic Voltage Scaling.* Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '00), 2000: p. 9-14.

10. T.D. Burd, T.A. Pering, A.J. Stratakos, R.W. Brodersen, *A dynamic voltage scaled microprocessor system.* IEEE Journal of Solid-State Circuits, 2000. 35 (11): p. 1571-1580.

11. R. Min, T. Furrer, A. Chandrakasan, *Dynamic voltage scaling techniques for distributed microsensor networks.* Proceedings. IEEE Computer Society Workshop on VLSI, 2000: p. 43 -46.

12. T. Ishihara , and H. Yasuura, *Voltage scheduling problem for dynamically variable voltage processors.* Proceedings 1998 international symposium on Low power electronics and design, 1998: p. 197-202.

13. Bennett, S., *Real-time computer control: an introduction*. 2nd ed. Series in Systems and Control Engineering, ed. M.J. Grimble. 1994, London: Prentice Hall International.

14. Joch, A., *Real-Time Operating Systems*, in *Computerworld*. 2001.

15. Liu, J.W.-S., *Real-time systems*. 2000: Prentice Hall.

16. C. L. Liu, and J. W. Layland, *Scheduling Algorithms for Multiprogramming in a hard real time environment.* Journal of the Association for Computing Machinery, 1973. v.20, n.1: p. 44-61.

17. D. B. Stewart and P. K. Khosla, *Real-time scheduling of sensor-based control systems*, in *Real-Time Programming*, W. Halang and K. Ramamritham, Editor. 1992, Pergamon Press. p. 139-144.

18. Wescott, T., *PID without PhD*, in *Embedded Systems Programming*. 2002.

19. M. Weiser, B. Welch, A. Demers, and S. Shenker, *Scheduling for reduced CPU energy.* Preceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI '94), 1994: p. 13-23.

20. T. Pering, T.Burd, and R. Brodersen, *The simulation and evaluation of dynamic voltage scaling algorithms.* 1998: p. 76-81.

21. L.H. Chandrasena, P. Chandrasena, M.J. Liebelt, *An energy efficient rate selection algorithm for voltage quantized dynamic voltage scaling.* Proceedings of the International Symposium on System Synthesis, 2001: p. 124 -129.

22. D. Grunwald, P. Levis, C.B.M. III, M. Neufeld, and K.I. Farkas, *Policies for dynamic clock scheduling.* Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000), 2000: p. 73-86.

23. F. Yao, A. Demers, and S. Shenker, *A scheduling model for reduced CPU Energy.* IEEE Annual foundations of computer science, 1995: p. 374-382.

24. T. Pering, T.Burd, R. Brodersen, *Voltage scheduling in the IpARM microprocessor system.* Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED '00), 2000: p. 96-101.

25. Donghwan Son, Chansu Yu, Heung-Nam Kim, *Dynamic voltage scaling on MPEG decoding.* Proceedings of the Eighth International Conference on Parallel and Distributed Systems (ICPADS 2001), 2001: p. 633- 640.

26. O.Y.-H. Leung, Chi-Ying Tsui, R.S.-K. Cheng, *Reducing power consumption of turbo-code decoder using adaptive iteration with variable supply voltage.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2001. 9(1).

27. Y. Shin , and K. Choi, *Power conscious fixed priority scheduling for hard real-time systems*. in *Proceedings of the 36th ACM/IEEE conference on Design automation conference*. 1999.

28. O.Y.-H. Leung, Chung-Wai Yue, Chi-Ying Tsui, R.S. Cheng, *Reducing power consumption of turbo code decoder using adaptive iteration with variable supply voltage.* Proceedings of the 1999 International Symposium on Low Power Electronics and Design, 1999: p. 36-41.

29. P.Pillai and K.G. Shin, *Real-time dynamic voltage scaling for low-power embedded operating systems.* Proceedings of the 18th Symposium on Operating Systems Principles (SOSP 2001), 2001: p. 89-102.

30. Woonseok Kim, Jihong Kim, and Sang Lyul Min, *A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis.* Proceedings of the Design Automation and Test in Europe (DATE 2002), 2002: p. 788-794.

31. H. Aydin, R. Melhem, D. Mosse and P.M. Alvarez, *Determining optimal processor speeds for periodic Real-time tasks with different power characteristics.* Proceedings of the 13th EuroMicro Conference on Real-Time Systems (ECRTS'01), 2001.

32. V. Swaminathan and K. Chakrabarty, *Real-time task scheduling for energy-aware embedded systems.* to appear in Proc. IEEE Real-Time Systems Symp. (Work-in-Progress Session), 2000.

33. V. Swaminathan, C.B. Schweizer, K. Chakrabarty, A.A. Patel, *Experiences in implementing an energy-driven task scheduler in RT-Linux.* Proceedings of

the IEEE Real-Time and Embedded Technology and Applications Symposium, 2002: p. 229-238.

34. S.M. Martin, K. Flautner, T. Mudge, D. Blaauw, *Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads.* IEEE/ACM International Conference on Computer Aided Design (ICCAD 2002), 2002: p. 721-725.

35. Lin Yuan and Gang Qu, *Design space exploration for energy-efficient secure sensor network.* 2002: p. 88-97.

36. Eui-Young Chung, L. Benini, and G. De Micheli, *Contents provider-assisted dynamic voltage scaling for low energy multimedia applications.* Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED '02), 2002: p. 42-47.

37. F. Gilbert, A. Worm, and N. Wehn, *Low power implementation of a turbo-decoder on programmable architectures.* Proceedings of the ASP-DAC 2001 Design Automation Conference, 2001: p. 400-403.

38. A. Sinha and A.P. Chandrakasan, *Energy efficient real-time scheduling.* IEEE/ACM International Conference on Computer Aided Design (ICCAD 2001), 2001: p. 458-463.

39. T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. de Micheli, *Dynamic voltage scaling and power management for portable systems.* Proceedings of the Design Automation Conference, 2001: p. 524-529.

40. Qu, G., *What is the limit of energy saving by dynamic voltage scaling?* IEEE/ACM International Conference on Computer Aided Design (ICCAD 2001), 2001: p. 560-563.

41. N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler, *Hybrid global/local search strategies for dynamic voltage scaling in embedded multiprocessors.* Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES'01), 2001: p. 243-248.

42. Jiong Luo, N.K. Jha, *Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems.* IEEE/ACM International Conference on Computer Aided Design (ICCAD-2000), 2000: p. 357-364.

43. Jha, N.K., *Low power system scheduling and synthesis.* IEEE/ACM International Conference on Computer Aided Design (ICCAD 2001), 2001: p. 259-263.

44. Sunghyun Lee, Sungjoo Yoo, Kiyoung Choi, *An intra-task dynamic voltage scaling method for soc design with hierarchical fsm and synchronous dataf low model.* Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED '02), 2002: p. 84-87.

45. A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, A. Nicolau, *Profile-based dynamic voltage scheduling using program checkpoints.* Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2002: p. 168-175.

46. I. Hong, M. Potkonjak, and M.B. Srivastava, *On-line Scheduling of Hard Real-time Tasks on Variable Voltage Processor.* Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design, 1998: p. 653-656.

47. K. Flautner , S. Reinhardt , and T. Mudge, *Automatic performance setting for dynamic voltage scaling.* Proceedings of the seventh annual international conference on Mobile computing and networking, 2001: p. 260-271.

48. C. Lee, M. Potkonjak and W. Mangione-Smith, *MediaBench: A tool for evaluating and synthesizing multimedia and communications systems.* In Proc. 30th Annual International Symposium on Microarchitecture (MICRO'97), Research Triangle Park NC, 1997: p. 330-335.

49. K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, *The performance and energy consumption of three embedded real-time operating systems.* Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001), 2001: p. 203-210.

50. Labrosse, J.J., *MicroC/OS-II: The Real-Time Kernel*. 1999, Lawrence, KS: R&D Books (Miller Freeman, Inc.).

51. J. Scott, L. H. Lee, J. Arends, and B. Moyer, *Designing the Low-Power M-CORE Architecture.* Proceedings of the IEEE Power Driven Microarchitecture Workshop, 1998: p. 145-150.

52. Y. Liu, A. K. Mok, *An Integrated Approach for Applying Dynamic Voltage Scaling to Hard Real-Time Systems.* Proceedings of the 9th IEEE Real-Time ad Embedded Technology and Applications Symposium (RTAS'03), 2003.

53. N. AbouGhazaleh, D. Mosse', B. Childers, F. Melhem, and M. Craven, *Collaboratice Operating System and Compiler Power Management for Real-Time Applications.* Proceedings of the 9th IEEE Real-Time ad Embedded Technology and Applications Symposium (RTAS'03), 2003.