2007

DYNAMIC VOLTAGE SCALING FOR PRIORITY-DRIVEN SCHEDULED DISTRIBUTED REAL-TIME SYSTEMS

Chenxing Wang
University of Kentucky, cwang0@engr.uky.edu

ABSTRACT OF DISSERTATION

Chenxing Wang

The Graduate School

University of Kentucky

2007

DYNAMIC VOLTAGE SCALING FOR PRIORITY-DRIVEN SCHEDULED DISTRIBUTED REAL-TIME SYSTEMS

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements of the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Chenxing Wang
Lexington, Kentucky

Co-Directors: Dr. Henry G. Dietz, Professor of Department of Electrical and Computer Engineering
and Dr. William R. Dieter, Professor of Department of Electrical and Computer Engineering

Lexington, Kentucky

2007

ABSTRACT OF DISSERTATION

DYNAMIC VOLTAGE SCALING FOR PRIORITY-DRIVEN SCHEDULED DISTRIBUTED REAL-TIME SYSTEMS

Energy consumption is increasingly affecting battery life and cooling for real-time systems. Dynamic Voltage and frequency Scaling (DVS) has been shown to substantially reduce the energy consumption of uniprocessor real-time systems. It is worthwhile to extend the efficient DVS scheduling algorithms to distributed system with dependent tasks.

The dissertation describes how to extend several effective uniprocessor DVS scheduling algorithms to distributed system with dependent task set. Task assignment and deadline assignment heuristics are proposed and compared with existing heuristics concerning energy-conserving performance. An admission test and a deadline computation algorithm are presented in the dissertation for dynamic task set to accept the arriving task in a DVS scheduled real-time system.

Simulations show that an effective distributed DVS scheduling is capable of saving as much as 89 % of energy that would be consumed without using DVS scheduling. It is also shown that task assignment and deadline assignment affect the energy-conserving performance of DVS scheduling algorithms. For some aggressive DVS scheduling algorithms, however, the effect of task assignment is negligible. The admission test accept over 80% of tasks that can be accepted by a non-DVS scheduler to a DVS scheduled real-time system.

KEYWORDS: Distributed Real-time System, Scheduling, Dynamic Voltage Scaling, Task and Deadline Assignment, Admission Test

Chenxing Wang

2007

DYNAMIC VOLTAGE SCALING FOR PRIORITY-DRIVEN SCHEDULED DISTRIBUTED REAL-TIME SYSTEMS

By

Chenxing Wang

Henry G. Dietz

Co-Director of Dissertation

William R. Dieter

Co-Director of Dissertation

Yuming Zhang

Director of Graduate Studies

2007

Date

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Master's and Doctor's degrees and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part requires also the consent of the Dean of the Graduate School of the University of Kentucky.

A library which borrows this dissertation for use by its patrons is expected to secure the signature of each user.

Name	Date

DISSERTATION

Chenxing Wang

The Graduate School

University of Kentucky

2007

DYNAMIC VOLTAGE SCALING FOR PRIORITY-DRIVEN SCHEDULED DISTRIBUTED REAL-TIME SYSTEMS

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements of the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Chenxing Wang
Lexington, Kentucky

Co-Directors: Dr. Henry G. Dietz, Professor of Department of Electrical and Computer Engineering
and Dr. William R. Dieter, Professor of Department of Electrical and Computer Engineering

Lexington, Kentucky

2007

(DEDICATION)

ACKNOWLEDGMENT

I wish to express my profound sense of gratitude to my advisor, Dr. William R. Dieter, for his guidance, support and encouragement throughout the course of the research. He greatly helped me grow in the academic research, making a diligent effort to lead me into the real-time computing world. By his example of hard work and his pursuit of excellence, he showed me the essentials of being a successful researcher.

I am indebted to Dr. Henry G. Dietz, the Chair of committee. He provided precious and instructive comments and evaluation of my research work, allowing me to complete this dissertation on schedule. Next, I wish to thank the complete Dissertation Committee, and the outside reader, respectively: Dr. James Lumpp, Dr. D. Manivannan, and Dr. H.S. Tzou. Each individual provided insights that guided and challenged my thinking, substantially improving the finished product.

In addition to the technical and instrumental assistance above, I received equally important assistance from family and friends. My husband, Jing, has gone through the whole process of writing a Ph.D. thesis with me. He sacrificed his time to give me comfort and let me concentrate on the work. Without his love, care and support, my finishing the thesis would never have been possible. To him I will always be grateful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: INTRODUCTION

The correctness of a hard real-time (RT) systems depends on its timeliness. In other words, deadlines of tasks in a hard real-time system have to be guaranteed. Periodic tasks are the primary task type running on the real-time systems. A periodic task releases at a constant rate and has a deadline indicating the maximum allowed time to complete. To ensure the timeliness of the system, tasks have to be executed in an order such that no deadline misses. The execution order of tasks is called a schedule. Static and priority-driven scheduling are the major categories of real-time scheduling approaches.

Static scheduling schedules the tasks offline and stores the static schedule for use at run time. Static scheduling has the minimized scheduling overhead during run-time [22]. However, static scheduling does not handle dynamically changing task sets well. For example, static schedules do not allow job release times to vary, as is the case if an outside event triggers a job release. Adding new tasks is difficult because the entire schedule must be recomputed, which often requires an expensive offline algorithm.

Priority-driven scheduling executes the task with highest current priority at run time. Among priority-driven scheduling algorithms, the Earlist-Deadline-First (EDF) is one of the most widely researched priority-driven scheduling. EDF algorithm gives the job with the soonest deadline the highest priority, allows the system to run at full utilization when tasks' deadlines equal their periods [21].

Many embedded systems have dynamic load, either due to events vary in environment, variations in workload, or adding and removing tasks from the system. For example, in a target tracking application, a new task may be introduced to maintain the track as the tracked object moves. Interfacing with the real world implies timing constraints for sampling sensors and/or controlling actuators. Thus, the system needs a real-time scheduling algorithm capable of guaranteeing that all tasks meet their deadlines. The priority-driven scheduling algorithms are better suited to applications with these dynamic properties. An arriving periodic task may be scheduled by priority-driven scheduling algorithms immediately if it passes a simple admission test because there is no need to recompute a static schedule.

Most of real-time tasks are used in energy-constrained systems whose lifetime is determined by how long battery power lasts. The speed at which the processor runs in these systems is a major factor in how much power the system consumes. Processor speed depends on how much computational work is required. Systems that

can reduce processor speed can save energy when less computation is needed. A real-time scheduler reserves time for each job assuming it will execute for its entire worst case execution time. In many cases, however, jobs finish in much less than the worst case time. Dynamic Voltage and frequency Scaling (DVS) takes advantage of the technique of voltage scalable CPU and the variation in computational workload in real-time task to reduce overall energy consumption.

A real-time scheduler using DVS changes the processor speed at run time to more closely match the amount computation required, while still guaranteeing that all jobs complete by their deadlines, even if every job requires its worst case execution time.

A number of DVS algorithms have shown significant energy savings when scheduling real-time jobs [14, 32, 31, 24, 36, 20, 18]. For hard real-time systems based on Earliest Deadline First (EDF), the Lookahead EDF (LAEDF) [31] and Feedback EDF algorithms [14, 47, 48, 49] are two of the top performers, while Static EDF and Cycle Conserving EDF (CCEDF) still save power , but have less runtime overhead. Several comparisons of real-time DVS algorithms have shown LAEDF and Feedback EDF produce close to optimal power savings, with Feedback EDF typically reduces more power consumption than LAEDF [14, 49].

Real-time applications, such as wireless sensor networks, air traffic control, and battle field surveillance, require distributed real-time systems. The scheduler for distributed system can be global or partitioned. The global scheduler maintains a global task queue to make scheduling decision for tasks waiting to be executed. The tasks migration is required in global scheduling. For some applications, real-time tasks can not be move around within the system either because of the expensive cost of context transmission or the physical limitation. The partitioned scheduling is used instead in the systems that task migration is prohibited. In partitioned distributed system, tasks are assigned to processor in system initialization step. There is a local scheduler on each of the processors within the system making scheduling decisions on the tasks assigned to it.

As with uniprocessor real-time systems, distributed real-time systems are usually energy constrained. However, DVS techniques are not well developed for such systems because of their complexity. Most existing DVS algorithms are based on static scheduling methods. There is ample space for research in DVS algorithms for priority-driven scheduled distributed real-time systems. The DVS algorithms for uniprocessor scheduling methods can be developed for partitioned distributed system, in which tasks are dependent on each other.

In practice, some tasks running on distributed real-time systems are related. These

tasks fulfill a system function when executed in order. That is, they have precedence constraints. Task graphs are commonly used to model related tasks in the system. There is a special case when each task in the task graph has at most one predecessor and one successor, which forms a task chain. In the real world, a wide range of real-time applications can be covered by the model of task chain, or *end-to-end task*.

This dissertation focuses on a suite of distributed DVS-EDF scheduling algorithms that save considerable energy consumption and are able to react to changes in task set, online task assignment and deadline assignment algorithms facilitating DVS-EDF scheduling algorithms in reducing the system energy consumption for the partitioned distributed real-time system with end-to-end task sets.

Most of the existing energy-aware scheduling approaches for distributed real-time systems are based on static scheduling. To explore the energy conservation ability of priority-driven scheduling, this dissertation proposes DVS scheduling algorithms for EDF scheduled distributed system based on uniprocessor DVS techniques.

The existing task assignment approaches for end-to-end task on distributed real-time systems focus on system's schedulability and total communication cost [22]. Most of these algorithms, such as genetic algorithm [16, 39, 29], and integer linear programing [22, 19], are only useful for off-line task assignment because of their complex nature. In fact, the system energy consumption is affected by how a set of tasks are assigned to the processors in the system. Moreover, dynamic task sets with tasks arriving and leaving the system, require an online task assignment algorithm to admit and schedule each new task. The online energy-aware task assignment approach discussed in this dissertation offers a way to assign task on the fly, while taking energy consumption into consideration.

To schedule subtasks in end-to-end task with a common deadline using EDF scheduling, each subtask has to be assigned its own deadline. The dissertation discusses and simulates the existing deadline assignment approaches to reveal the relationship between deadline assignment and system energy consumption. An energy-aware deadline assignment algorithm is proposed in the dissertation in order to facilitate the energy conservation of DVS scheduling.

EDF scheduling handles dynamic task set well because of its ability to admit new task online with very low computation overhead. However, the simple EDF admission test can not be used for real-time system with DVS-EDF applied. This dissertation proposed an approach to handle dynamic task set on a real-time system scheduled by LAEDF scheduler.

An overview of background and related work is given in Chapter 2. Some DVS

scheduling algorithms for uniprocessor real-time system are discussed and compared. Priority-driven scheduling for distributed with end-to-end task set is introduced in this chapter, which serves as basics of the further discussion of distributed DVS algorithms. Chapter 3 describes the system model used to design and simulate the algorithms. Distributed DVS-EDF scheduling algorithms are discussed in Chapter 4. The task assignment algorithms and deadline assignment heuristics are described in Chapter 5. Two admission test algorithms for dynamic task sets scheduled with DVS-EDF in uniprocessor real-time system are proposed and discussed in Chapter 6. In Chapter 7, four groups of simulation results are described for task assignment, deadline assignment, distributed DVS-EDF, and amission tests for dynamic task set. Conclusions are drawn in Chapter 8 based on the discussion and simulation results of all the algorithms in the dissertation followed by a brief description of future work direction.

Chapter 2: BACKGROUND AND RELATED WORK

In real-time systems, a *periodic task* is the most common task type. A periodic task releases at a constant rate, its *period*; has a maximum allowed time to complete, its *relative deadline* and a maximum processor time to finish the task, known as *worst-case execution time (WCET)*. One instance released from a task is called a *job* of the task. The *absolute deadline* of a job is the time by which the job has to be finished, which is the sum of release time of the job and the tasks *relative deadline*. Besides periodic tasks, there are *aperiodic tasks* and *sporadic tasks* in real-time systems. They are released in a random manor. The difference between them is, an aperiodic task has no deadline, but a sporadic task has an absolute deadline.

The correctness of hard real-time systems depends on *timeliness*; each job must finish before its deadline. To ensure the timeliness of the system, tasks have to be executed in an order such that no deadline is missed. The execution order of tasks is called *schedule*. The schedule is *feasible*, if no job misses a deadline when tasks are executed according to this schedule.

Static scheduling and priority-driven scheduling are two scheduling methods for RT systems. *Static scheduling* schedules tasks off-line and stores the static schedule for use at run time. Static scheduling has minimal scheduling overhead during run time [22] because all scheduling decisions are fixed. *Priority-driven* scheduling schedules the task with highest priority at each scheduling decision point at run time. Priority-driven scheduling algorithms may use *fixed-priority* or *dynamic-priority*. In fixed-priority systems, each task has its fixed priority level assigned before the task is added to the system, for example, the *Rate Monotonic (RM)* algorithm assigns each task a priority proportional to its period. Dynamic-priority schedulers decide which task should run first based on dynamic characteristics, such as absolute deadlines. Tasks with higher priorities always run first when they are ready to run. The *Earliest Deadline First (EDF)* scheduling algorithm is widely known scheduling approach in which the task with the earliest deadline is given highest priority.

A *task* is a sequence of related jobs. For real-time systems, we denote the i^{th} periodic task T_i. p_i is its period, and D_i is its deadline. C_i is the worst-case execution time of the task, while c_i is the actual execution time (AET) of one job in task T_i. $J_{i,k}$ will be used to denote the k^{th} job (instance) of the i^{th} task. *Utilization* of the task, the percent of the tasks period it spends on executing, is denoted as u_i. The slow-down factor α describes the effect of frequency scaling on the execution speed.

Slack time, time remaining in excess of allotted time when a job is finished, always

exists in practical real-time systems. Slack time can be static if the deadline is longer than job's WCET, or dynamic if the WCET is longer than the actual execution time of a job. In practical real-time systems, the actual execution time changes from job to job within the same task. Variation in execution time can be caused by cache misses, different path in program flow, and different number of iterations in loops etc. To ensure the feasibility of the system, the maximum or worst case execution time is used when scheduling the task. *Dynamic slack time* is generated when job runs less than the maximum execution time.

2.1 DVS for Uniprocessor Real-Time System

DVS scheduling for real-time systems takes advantage of voltage scalable processors. By lowering the supply voltage of a voltage scalable processor, the power consumption of the real-time system is lowered. The relationship between processor's power consumption and its voltage supply, and some uniprocessor DVS scheduling algorithms are described in the rest of this section.

2.1.1 Power Model of Voltage Scalable Processor

The DVS technique for voltage scalable processors has seen its wide applications in industry to lower the system's energy consumption. Lowering a processor's supply voltage reduces the power consumed by the processor as well as the processor's speed. For CMOS devices, the relationship between the power consumption and the device's voltage supply is modeled by a nonlinear equation [4].

$$P = C_d V_{dd}^2 f + C_s V_{dd} I_{leakage} \tag{2.1}$$

And the device's speed (frequency) is related to the supply voltage.

$$f \propto \frac{(V_{dd} - V_{th})^2}{V_{dd}} \tag{2.2}$$

Where V_{dd} is the supply voltage, f is the device's frequency, C_d and C_s are dynamic constant and static constant respectively. $I_{leakage}$ is the leakage current, and V_{th} is the threshold voltage of the device, which is small when compared with supply voltage.

The first term in Equation 2.1 stands for the dynamic power consumption that is caused by switching of CMOS circuits. While the second term is modeled for static power consumed when the leakage current flowing through the transistors. Until recently, static power was substantially smaller than switching power. However, by

lowering voltage thresholds to increase speed in CMOS designs, static power has become comparable to dynamic power in high speed CMOS devices.

DVS technique reduces the dynamic power consumption by slowing down the execution of real-time tasks while guarantee their timeliness. The lower processor operation speed requires lower supply voltage. A decrease in the supply voltage results in approximately a cubic reduction in dynamic power consumption according to the first term of Equation 2.1. Although DVS techniques are used to reduce the dynamic power consumption, static power consumption caused by leakage current can only be reduced by putting the system to sleep.

2.1.2 DVS-EDF for Uniprocessor Real-Time System

DVS scheduling algorithms for real-time systems can be categorized as static or dynamic DVS according to the type of scheduler that they work for. The static DVS algorithm scales the execution speed of tasks by a constant factor. When making the decision on the task's execution speed, static DVS algorithms assume the task requires its worst case execution time. The dynamic slack time exists in many applications of real-time tasks. Most of the dynamic DVS algorithms can take advantage of dynamic slack time when trying to reduce the task's execution speed. DVS (DVS-EDF) for uniprocessor system is developed based on the widely used EDF scheduling approach.

Static Speed EDF

The Static speed EDF (SEDF) algorithm chooses the lowest possible processor frequency, f_α, that can be used to run tasks without a missing deadline [43]. For EDF with system task's deadline equal to its period, the feasibility of a task set is determined by the total utilization of the real-time system. If the utilization is less than or equal to 1, the system can be feasibly scheduled by EDF scheduling. If the utilization is less than one, there exists static slack time that can be used to slow down the CPU by the rate of

$$\alpha = \sum_{T} \frac{C_i}{p_i} \tag{2.3}$$

And thus,

$$f_\alpha = \alpha f_{ref} \tag{2.4}$$

Where T is task set scheduled feasibly in the system. f_{ref} is the highest processor frequency.

The frequency of the system is set to f_α during initialization and kept constant thereafter. There is no online overhead in Static speed EDF, since the speed is decided offline. However SEDF algorithm is not efficient because it does not take the advantage of the system's dynamic slack time. This algorithm is usually combined with other DVS algorithms to obtain better energy conservation.

Stretching to Next-Task-Arrival

Stretching to Next-Task-Arrival (NTA) [2] tries to scale the processor's frequency dynamically based on the next task arrival time. Assume the current job, J, is released at time t. J stretches its execution time so that it finishes just before the next job arrives or just before its deadline, whichever comes first. If the next job arrives before the current job can finish its WCET execution, the job has to be executed at the full speed. Equation 2.5 can be used to select processor speed when the job is released.

$$\alpha = \begin{cases} C_i/(d_i - t), & NTA > d_i \\ C_i/(NTA - t), & C_i + t < NTA \leq d_i \\ 1, & NTA \leq C_i + t \end{cases} \tag{2.5}$$

Where C_i and d_i are WCET and absolute deadline of current task, and NTA is the arrival time of the next task. This algorithm is simple and easy to implement, but the slack time estimation is too simple to get a good approximation of the dynamic slack time. The frequency can change with every task switch and quite often the system must run at full speed. Also, Equation 2.5 must be evaluated at every context switch.

Cycle-Conserving EDF

Like the Static Speed EDF algorithm, cycle-conserving EDF (CCEDF) [31] also uses utilization updating to scale the CPU speed. The difference is that dynamic slack time as well as static slack time is exploited to update current utilization of the real-time system.

The dynamic slack time is caused by the difference between the worst-case execution time, C_i, and the actual execution time c_i. CcEDF updates the total utilization on the fly by using the actual execution time for completed tasks and the worst-case execution time for those just released. That is,

$$\alpha = U = \sum_{T_k \in completed\ tasks} \frac{c_k}{p_k} + \sum_{T_l \in released\ tasks} \frac{C_l}{p_l} \tag{2.6}$$

$$f = \alpha f_{ref} \tag{2.7}$$

Where α is slow-down factor calculated for current executing task T_i. Equation 2.6 has to be evaluated each time a task completes its current job or another job is released from a task. Since the system utilization can be updated based on the previous calculation, CCEDF has low online overhead with computational complexity of $O(1)$.

Lookahead EDF

The Lookahead EDF (LAEDF) algorithm is more aggressive in exploiting the dynamic slack when trying to scale down the processor speed [31]. LAEDF reduces the amount of work the processor must do by deferring as much work as possible until after the current job's deadline, then slowing down the processor until it is just fast enough to run the undeferrable work, and finishing just before the deadline.

Greater energy conservation is expected when using LAEDF due to its more sophisticated dynamic slack estimation. Better estimation, however, causes higher online overhead. LAEDF has a linear computational complexity of $O(n)$, where n is the total number of tasks that is running on the system.

Feedback EDF

Feedback EDF takes advantage of dynamic slack due to the jobs that require less than their worst-case execution time [47, 14, 49, 48, 50]. A job, $J_{i,k}$ in task T_k has its WCET, C_k, divided into two parts such that,

$$C_k = C_A + C_B \tag{2.8}$$

Where C_A is an estimation of current job's execution time (e.g. average execution time of prior jobs), which is assumed to be the actual execution of currently released job, $J_{i,k}$. The processor speed can thus be scaled down by

$$\alpha_k = \frac{C_A}{C_A + S_k}, \tag{2.9}$$

where S_k is the accumulated slack time passed from prior completed jobs. S_k

contains two major parts, slack time generated by prior jobs and that accumulated from the idle task. The idle task is a periodic task with zero execution time and its period equals to the shortest period among tasks' periods on the system. If $J_{i,k}$'s actual execution time is greater than the estimation, C_A, the full processor speed of f_{ref} has to be applied to run the second part of the job in order to catch the deadline.

A PID execution time predictor, borrows the concept of PID control to estimate the future actual execution time from the past execution times. It can be used to make a better estimation for some specified real-time applications. However, there is no guarantee that running the second part of job can be avoided.

Feedback EDF performs better than CCEDF and LAEDF when actual execution time is less than or equal to estimated execution time. It exploits not only inter-task dynamic slack time but also tries to utilize the dynamic slack time generated by the released job itself in advance. However, if there exists a nonzero C_B, full speed may have to be applied to the processor, which in practice may lower the battery efficiency and reduce the battery life due to higher current draw. The online overhead is expensive in this algorithm, especially when PID predictor is used.

2.1.3 Comparison of Uniprocessor DVS-EDF Algorithms

To quantitatively compare DVS-EDF algorithms, each DVS algorithms is applied to a real-time system with 20 independent preemptable periodic tasks. In order to make a fair comparison, two different patterns of actual execution time are used for task generation: uniform distributed and Gaussian distributed. The actual execution time of each task is randomly distributed among 0 to task's WCET.

The simulation results are given in Figure 2.1 and Figure 2.2, respectively. In each of these figures, y-axis is the system energy consumption using DVS-EDF algorithms normalized to the energy consumption using an EDF scheduling. The X-axis is the utilization of the real-time system.

CCEDF and LAEDF have a very good performance in terms of energy conservation. Static Speed EDF performs well when the load of system is light. When the system load is increasing, the energy consumption for Static Speed EDF increases quickly.

The Feedback EDF performs best among the DVS-EDF algorithms with both uniformly distributed and Gaussian distributed AET. The performance of this algorithm is affected by the pattern of task's actual execution time. Feedback EDF is capable of saving more energy with Gaussian distributed AET than that with uniformly dis-

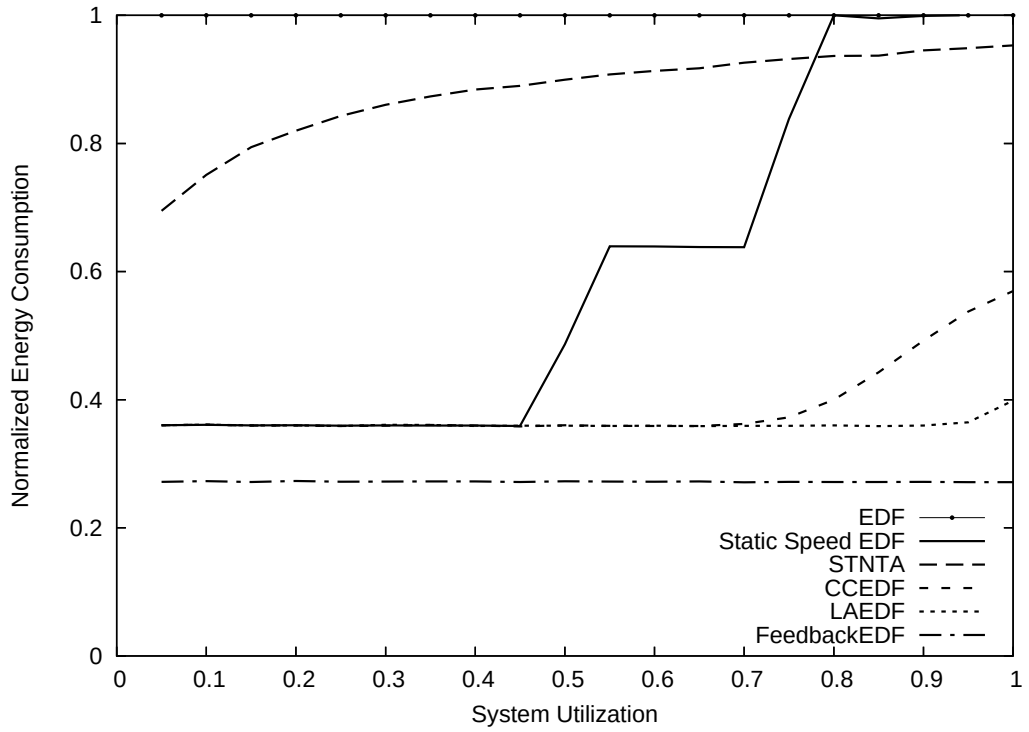


Figure 2.1: Normalized energy consumption with uniformly distributed AET

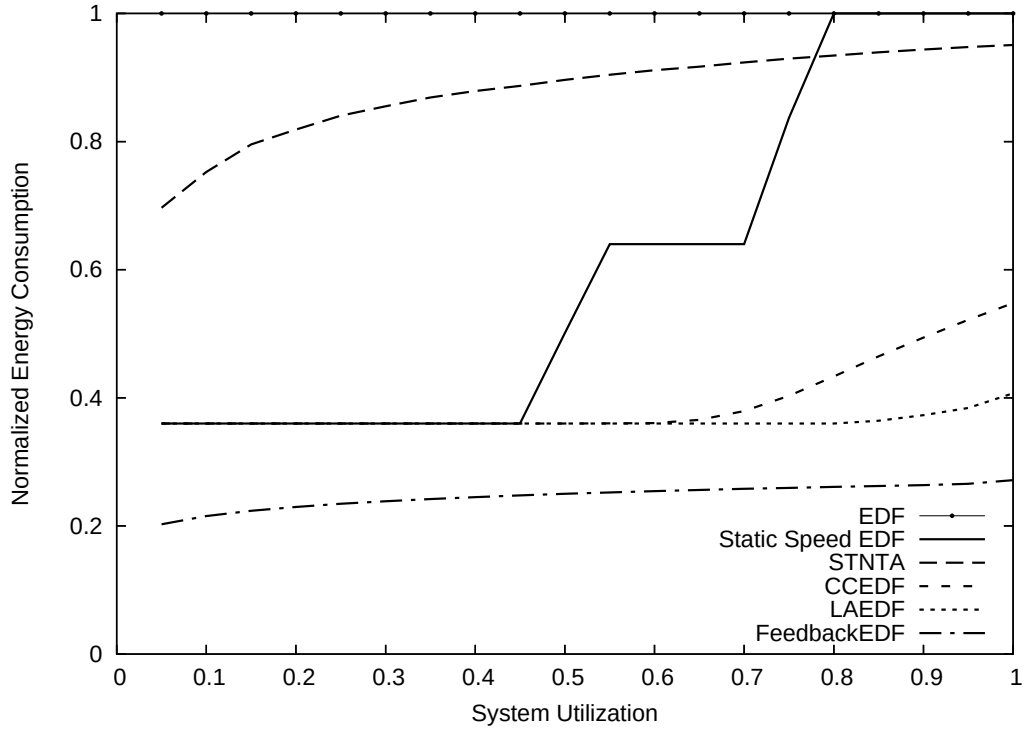


Figure 2.2: Normalized energy consumption with Gaussian distributed AET

tributed AET. This is because the system slack estimation of Feedback EDF predicts execution time based on the average of previous actual execution times. Even with the same average, the uniform distribution has a wider variance than the Gaussian distribution. The estimation is made on all the previous actual execution times.

2.2 End-to-End Scheduling for Distributed RT System

As introduced in Chapter 1, the scheduling of distributed real-time system can be classified into two major approaches, global scheduling and partitioning-based scheduling. The global scheduling requires task migration among processors, while partitioning-based scheduling statically assigned tasks onto each processor. To maintain the timeliness of the real-time system, the global scheduling requires low latency in communication between processors, which is not assume for most distributed systems. By assigning the tasks onto fixed processors, the partitioned distributed system greatly reduces the amount of data communicated between processors.

Tasks running on distributed real-time systems are usually related. These tasks fulfill a system function when executed in order. The dependency relation between these subtasks can be represented by a task graph. There is a special case when each task in the task graph has at most one predecessor and one successor, which forms a task chain. In the real world, a wide range of real-time applications can be modeled as a task chain. The task chain has a release time, *end-to-end release time*, and an *end-to-end deadline*, shared by each task in the chain. This task chain is called an *end-to-end task*. Each task within the chain is a *subtask* of that end-to-end task.

Subtasks may run on different processors within a distributed system. The basic steps of scheduling end-to-end tasks in distributed real-time systems are task assignment, deadline assignment, task synchronization and task scheduling.

2.2.1 Task Assignment and Deadline Assignment

Task assignment is the first step when scheduling end-to-end task set. Off-line task assignment approaches can be formulated as a constrained optimization problem as solved with techniques like integer linear programming problem. The selection of costs is based on the purpose of real-time system design. The communication cost between processors is frequently considered when assigning the tasks to processors.

For homogeneous systems, the communication cost between two tasks on different processors depends on the volume of data exchanged and the bandwidth of the communication link [22]. To simplify the problem, it is assumed that the cost incurred

by communication link is the same across the system, which is always true when the system is connected by a broadcast network. Thus, the communication cost between two subtasks can be valued according to the volume of communication data. Along with the utilization constraints for each of processors, a cost function is formulated for the task assignment problem.

Though integer linear programming can find an optimal solution, it and other optimization algorithms are too time consuming to use online. Fortunately, simple and fast bin-packing heuristics, such as Worst-Fit, Best-Fit, and First-Fit, work well for task assignment. These algorithms have low computation overhead and are used in systems with dynamic task sets, where tasks arrive and leave the system while it is running.

Subtasks within an end-to-end task share an end-to-end deadline. In order to feasibly schedule the subtasks that have been assigned to a processor using a priority-based scheduling algorithm, a local deadline has to be assigned to each of the subtask based on the end-to-end deadline. The method used to decide the local deadline for subtasks is *deadline assignment*. Two existing deadline assignment algorithms are proportional deadline (PD) and normalized proportional deadline (NPD) [22]. PD assigns the deadline proportional to the subtask's worst-case execution time. While NPD assigns the subtask's deadline according to its worst-case execution time as well as the workload on each processor. A detailed discussion of PD and NPD is given in Chapter 5, along with their effect on power consumption.

2.2.2 Interprocessor Synchronization

The task model in distributed real-time systems differs from one in uniprocessor systems in two ways, subtasks have dependencies and share an end-to-end deadline. Deadline assignment breaks the end-to-end deadline into deadlines of subtasks. Task synchronization is used to maintain the dependence between subtasks, and can be characterized as greedy or non-greedy. *Greedy synchronization* allows a task to be released as soon as all of its predecessors are completed. Though it allows for higher average throughput, tasks can be released more often than normal periodic tasks would be released, jeopardizing schedulability. *Non-greedy synchronization*, on the other hand, delays task release when necessary to preserve periodic behavior. Non-greedy synchronization can perserve schedulability when the right scheduling algorithm is used.

Among the various non-greedy synchronization protocols, the release-guard (RG)

protocol performs best [22]. The RG protocol makes sure that the release intervals of any subtask are never less than the period of the subtask, and keeps completion-time jitter small. Unlike other non-greedy protocols that synchronize the processors based on the subtasks upper bound of response time, the RG protocol uses the information of the subtasks last release time and its period when deciding the next release time for that subtask. These characteristics make the RG protocol ideal for interprocessor synchronization in priority-driven scheduled distributed real-time systems.

After task assignment and task model transformation, tasks can be scheduled on each processor using uniprocessor scheduling methods, either dynamic-priority or fixed-priority scheduling, with task synchronization between processors.

2.3 DVS for Distributed Real-Time Systems

Most of DVS for distributed real-time system are based on static scheduling [33, 46, 26, 25, 13]. Some of these DVS algorithms take advantage of task graph to get an energy-aware voltage schedule [33, 46]. Others try to statically schedule the tasks with minimized energy consumption by formulating and solving an optimization problem [37, 1, 5].

Only a few DVS algorithms have been proposed for priority-driven scheduled distributed real-time system [1, 5, 28]. One proposed algorithm is based on system synthesis [1, 5]. Another has applied a simple DVS algorithm for priority-driven scheduled distributed real-time systems [28].

Static Power Management for Distributed Real-Time System

The Static Power Management (SPM) algorithm [33] exploits system static slack time to lower system energy consumption. Three different variations on SPM distribute the static slack time among the tasks in different ways: greedy SPM (G-SPM), simple SPM (S-SPM), and SPM with parallelism (P-SPM).

G-SPM shifts the schedule toward the tasks deadline and allocates the entire static slack time to the first task on each processor. By slowing down the first task on each processor, the system energy is lowered. S-SPM differs from G-SPM in that S-SPM proportionally distributes the static slack to each task according to the worst-case execution time.

P-SPM is proposed based on the observation that more energy savings can be obtained by giving more slack to sections with higher parallelism. P-PSM takes the

degree of parallelism into consideration when allocating static slack time to different sections of a distributed schedule. P-SPM formulates the total energy consumption in terms of time intervals of different parallelism degree. By minimizing the total energy, lengths of time interval of all parallelism degree can be solved and used to slow down CPU speed.

Critical Path Analysis Algorithm

Critical path analysis algorithm (CPA) [46] statically extends the worst-case execution time of tasks scheduled by static scheduling in the distributed system to reduce the speed of processors through critical path analysis. The critical path in the task graph is defined as the path that has the minimum ratio as follows.

$$scale_j = \frac{t_{deadline} - t_{release}}{\sum_{T_i \in Path_j} C_i} \tag{2.10}$$

Where $Path_j$ is a task chain formed by a set of dependent tasks. $t_{deadline}$ is the absolute deadline of the last task in $Path_j$. While $t_{release}$ is the release time of first task in the path.

CPA scales WCET for all the tasks in the critical path by the scale ratio calculated by Equation 2.10. The tasks in the critical path are removed from the task graph and new deadlines are added to the graph to ensure the starting time of deleted tasks. Another critical path analysis can be done for the new task graph. The algorithm does the critical analysis iteratively until there is no task let in the task graph. The slow-down factor for processor can be determined by the ratio of task's WCET to its extended execution time.

Energy-Efficient Synthesis of Distributed RT System

Energy-efficient synthesis of distributed EDF [1, 5] algorithm works with independent task sets in the distributed real-time systems.

For a distributed system that has a set of independent periodic tasks, there might be more than one feasible schedule. Usually a linear programming problem has to be formulated to look for the optimal one results in minimized energy consumption. This algorithm formulates the constraints according to the time constraints of real-time tasks and EDF scheduling scheme. All the constraints are in terms of lengths of time interval and their corresponding processor speeds. Solving the processor speed for each time interval by minimizing the value of the cost function with constraints

is called *generalized distributed feasibility (GMF)* problem. The solution to GMF problem is an optimal feasible schedule for this distributed real-time system with a given task set.

The power consumed by m processors with computing capacity (processor frequency) of s would be proportional to

$$P(m, s) \propto ms^3, \tag{2.11}$$

s in the above equation dose not stand for the actual computing capacity of every processor. The actual value of s is subject to the equation below.

$$S_{sum} \leq ms - (m-1)S_{max}, \tag{2.12}$$

where S_{sum} is the total computing capacity of the system, and S_{max} is the computing capacity of the fastest processor in the system.

Minimizing Equation 2.11 subject to the GMF constraints including Equation 2.12, the problem becomes a nonlinear optimization problem. The solution of this optimization problem can obtain minimized energy consumption for this distributed system assuming all of the tasks execute with their worst-case execution time.

Power Variation DVS of Distributed Real-Time System

Power variation DVS algorithm [37] assumes that different processors in distributed system have different power profiles. This difference is taken into consideration when distributing the slack time among tasks. Power variation DVS algorithm is applied to a distributed real-time system with all the tasks been assigned to processors. This algorithm formulates an optimization problem to minimize the total energy E_Σ,

$$E_\Sigma = \sum_{T \in TaskSet} \frac{P_{max}(T) WCET(T)}{V_{max}^2(T) V_{dd}^2(T)}. \tag{2.13}$$

A set of $V_{dd}(T)$ for each of the task can be decided with real-time constraints and supply voltage range limits. P_{max} is the power consumed by the task when highest voltage, V_{max}, is supplied.

An energy gradient ΔE is introduced in this algorithm to solve above optimization problem using a hybrid global/local search strategy.

$$\Delta E_T = E_T(t) - E_T(t + \Delta t) \tag{2.14}$$

Where Δt is a time quantum.

The above four algorithms are applied to static-scheduled distributed real-time system. They make decisions on processor speed based on task's WCET. No dynamic slack time can be exploited by them either because of the nature of these algorithms and their expensive computational complexity.

Low-Power Distributed EDF

Low-Power Distributed EDF (LPDEDF) algorithm [28] is applied to a distributed real-time system scheduled by EDF. It assumes that the tasks have been properly assigned to the processors. The basic idea of LPDEDF is similar to stretching-to-NTA for uniprocessor real-time systems. LPDEDF assumes that the processors in the system can work in three different modes, idle mode, slow-down mode and full-speed mode.

When there is no active task on a particular processor in the system, the processor is set to idle mode till next task is activated. When there are more than one tasks ready to execute or the only activated task has a successor task, the processor runs at full speed till the current task completes. Otherwise the processor is slowed down so that the current task completes just at the time of next task arrival time or its deadline, whichever comes first.

As the stretching-to-NTA for uniprocessor real-time systems, LPDEDF algorithm is easy to implement with low online overhead, but not efficient enough in exploiting system dynamic slack time.

Chapter 3: SYSTEM MODEL

The dissertation assumes a distributed real-time system composed of a set of homogeneous microprocessors. Each processor has its own memory system and peripherals. The system is connected by a network, through which the processors exchange data and messages.

The set of processors, $\boldsymbol{P}=\{P_1, P_2, \ldots, P_m\}$, in the system are homogeneous in that they share a common architecture. To apply DVS technology, processors within the system have to be voltage scalable. The dissertation assumes that all the processors in the system support a same set of speeds, $\boldsymbol{F}=\{f_1, f_2, \ldots, f_n\}$. We refer to a speed adjustment, α, relative to the highest possible speed, f_{ref}. When $\alpha = 0.25$, for example, all jobs take four times longer to execute than at f_{ref}.

A speed/power table gives the average power consumed by the processor at each speed. We assume that execution time for each job is proportional to the processor speed regardless of the job, and that transition time to reach the desired operating speed is negligible compared to a task's execution time. The delay caused by the context switch on multitask systems is ignored or can be considered as part of worst-case execution time of real-time tasks.

Messages and data are transmitted through the network between processors in our system. Communication is used to synchronize processors and exchange data. We assume that the energy consumption caused by communication between processors is proportional to the number of bits that are transmitted.

The dissertation compares the energy consumption when the system is scheduled by different DVS-EDF algorithms. We take energy consumption of processors and the network into consideration. Energy consumption of memory systems is assumed to be included in processor power consumption. The system's energy consumption, E_{sys}, is composed of energy consumption of processors, E_{proc}, and that of the system communication, $E_{network}$.

$$E_{sys} = E_{proc} + E_{network} \tag{3.1}$$

A wide range of task sets running on real-time systems can be modeled as task chains. The task chain has an end-to-end release time and an end-to-end deadline shared by each task within the chain, as shown in Figure 3.1. This task chain is referred to as an *end-to-end task*. Each task within the chain is a *subtask* of that end-to-end task.

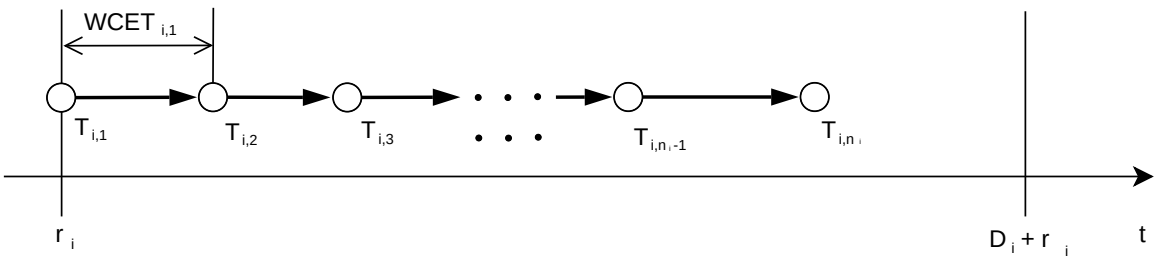


Figure 3.1: End-to-end task and its subtasks

We assume that there are n end-to-end tasks, $\boldsymbol{T}=\{T_1, T_2, \ldots T_n\}$, running on our distributed real-time system with m homogeneous processors, $\boldsymbol{P}=\{P_1, P_2, \ldots, P_m\}$. An end-to-end task, T_i, is composed of n_i preemptable subtasks, $T_{i,1}, \ldots T_{i,n_i}$.

The end-to-end task, T_i, is the *parent task* of its subtasks. The subtask, $T_{i,j}$, is called the *sibling task* of another subtask, $T_{i,k}$ ($j \neq k$). $T_{i,j}$ is the *predecessor (successor)* of $T_{i,k}$, if $j < k$ ($j > k$). Except for the first subtask, $T_{i,1}$, each subtask, $T_{i,j}$ ($2 < j \leq m$), must wait for its *immediate predecessor*, $T_{i,j-1}$, to complete before it can be released.

The first subtask in T_i is released periodically with the period of p_i and executes for time $C_{i,1}$ in the worst case. Each of the subtasks, $T_{i,j}$, is released with its WCET of $C_{i,j}$ and executes the actual execution time of $c_{i,j}$, $c_{i,j} \leq C_{i,j}$. The release time of the first subtask in $T_{i,1}$ is considered as the release time of T_i. The last subtask in T_i must complete within D_i time units from T_i being released. This deadline, D_i, referred to as the end-to-end deadline, must be met by the entire end-to-end task T_i. We assume that the deadline of the end-to-end task equals to its period. The relationship of end-to-end task and its subtasks are depicted in Figure 3.1.

A *job* is an instance of a subtask, which is released periodically from the subtask. $J_{i,j}^k$ is the k^{th} job released from the subtask $T_{i,j}$. Job $J_{i,j}^k$ is released at $r_{i,j}^k$ and must complete before its *absolute deadline*, $d_{i,j}^k$, which is $D_{i,j}$ time units from the job is released, i.e. $d_{i,j}^k = r_{i,j}^k + D_{i,j}$. To simplify the notation, when a job's release time and its absolute deadline are used to refer to a job independently, only a single subscript is used, e.g. J_i belongs to an unspecified task and has release time r_i and deadline d_i. Since we assume that the end-to-end task's deadline equals its period, one job in a subtask can be released no earlier than the completion of the previous job in the same subtask.

Scheduling end-to-end tasks requires first assigning tasks to processors, dividing the end-to-end deadline among the subtasks, synchronizing tasks, and scheduling the tasks. Subtasks of an end-to-end task may be assigned to different processors, but

once assigned to a processor subtasks do not migrate between the processors.

At run time tasks are synchronized with the Release Guard protocol [22]. With release guard, a subtask is released when its predecessor completes or one period after the previous job in the subtask was released, whichever comes later. The requirement to wait at least one period after the release of the previous job in the same subtask means that the subtasks on one processor behave like periodic tasks, allowing them to be scheduled using uniprocessor priority-driven scheduling algorithms. A side effect of deadline assignment is that subtasks behave like periodic tasks with deadline shorter than period. In addition, release guard causes jobs within a subtask have inter-release times that are sometimes longer than the task's period. Both of these conditions violate the assumptions of the most effective DVS scheduling algorithms, but do not usually change feasibility analysis for uniprocessor scheduling algorithms.

Chapter 4: DISTRIBUTED DVS-EDF SCHEDULING

This chapter formulates the problems that have to be solved when implementing uniprocessor DVS-EDF algorithms in distributed real-time systems. The solutions for each of the uniprocessor DVS-EDF scheduling algorithms are discussed. For several DVS-EDF algorithms, such as CCEDF, LAEDF and Feedback EDF, the detailed algorithms for distributed real-time systems are described and discussed in the following sections.

4.1 Problems in Implementing Distributed DVS-EDF

Recall that in Chapter 2 we discussed the transformation of end-to-end task model which allows the subtasks running on each of the processors in a distributed real-time system to be scheduled using uniprocessor scheduling algorithms. There are two main challenges when applying uniprocessor DVS algorithms to the end-to-end task model.

Some of the uniprocessor DVS-EDF algorithms make an assumption that the deadline and the period are equal, which is true for may uniprocessor real-time systems. For systems with deadlines equal to their periods, EDF scheduling algorithm can always generate a feasible schedule if and only if the system's utilization is less than or equal to one.

$$U_{sys} = \sum_{T} \frac{C_i}{p_i} \leq 1 \;\Leftrightarrow EDF \;\; schedulable \tag{4.1}$$

In the distributed real-time system, the subtask deadline assigned using deadline assignment algorithms, such as PD and NPD, is shorter than or equal to the subtask's period. Instead of system's utilization, system's density has to be used for system schedulability test. That is the system can be scheduled feasibly using EDF if the system's density is less than or equal to one.

$$\Delta_{sys} = \sum_{T} \frac{C_i}{min(p_i, D_i)} \leq 1 \;\Rightarrow\; EDF \; schedulable \tag{4.2}$$

Changes have to be made in order to handle the subtasks with a shorter deadline than period in the distributed real-time system.

Another challenge lies in that most of existing uniprocessor DVS-EDF algorithms assume independent real-time task set in the system. In the end-to-end task model, however, subtasks within an end-to-end task are precedence contrained. Each subtask except for the first one in an end-to-end task must wait for its predecessor to complete

before it can be released. With the release guard protocol, a subtask is released either one period after its last release time or after its predecessor completes, whichever comes later. The behavior of the release guard introduces release jitter to each of the subtasks running on a processor. Some uniprocessor DVS schedulers, such as LAEDF and Feedback EDF, can not handle the release jitter caused by the subtask dependency.

The Stretching-to-NTA is not affected by the above two problems for distributed real-time system. The LPDEDF for distributed system discussed in Chapter 2 shares the same basic idea with Stretching-to-NTA. Instead of three basic working modes for processors, full speed, low speed, and idle, assumed by LPDEDF, more specified processor working modes can be used by Stretching-to-NTA.

The detail of extension of other DVS-EDF algorithms list in Chapter 3 is discussed in following sections.

4.2 Distributed Static EDF

The extension of Static EDF (SEDF) to Distributed Static EDF (DSEDF) is straightforward [40]. When $D_i \leq p_i$, the utilization test for schedulability is replaced with the density test in Equation 4.2. Regardless of whether task relative deadlines are less than, equal to, or greater than their periods DSEDF will feasibly schedule them. By setting α to the smallest available value above α in Equation 4.3, the effective density remains as close to 1 as possible without exceeding it.

$$\alpha = \sum_{T} \frac{C_i}{min(p_i, D_i)} \tag{4.3}$$

4.3 Distributed CCEDF

The extension of CCEDF to Distributed CCEDF (DCCEDF) is similar to that of SEDF to DSEDF. DCCEDF will produce feasible schedules for tasks with deadlines shorter than their periods if density is substituted for utilization in CCEDF. That is, processor speed is set to the smallest α greater than Δ_{sys} in Equation 4.4. This algorithm is correct because a system is schedulable as long as its instantaneous density does not exceed 1 [22].

$$\alpha = \sum_{T_k \in completed\ tasks} \frac{c_k}{min(p_k, D_k)} + \sum_{T_l \in released\ tasks} \frac{C_l}{min(p_l, D_l)} \tag{4.4}$$

4.4 Distributed LAEDF

LAEDF is an effective energy saving DVS-EDF scheduling algorithm, as is distributed LAEDF. A detailed description of uniprocessor LAEDF will aid in understanding the DVS extensions to it.

4.4.1 Uniprocessor LAEDF

As we have discussed briefly in Chapter 2, LAEDF is a power-aware priority driven real-time scheduling algorithm, based on the EDF scheduling. Just like EDF, LAEDF gives the job with the soonest absolute deadline the highest priority. In addition, it scales the system speed as the job runs to dynamically reduce energy consumption.

LAEDF reduces the amount of work the processor must do by deferring as much work as possible until after the current job's deadline, then slowing down the processor until it is just fast enough to run the undeferable work, and finishing just before the deadline. To determine how much work can be deferred, LAEDF tracks how much work is left in each job, J_i, using C_left_i. C_left_i is set to C_i when J_i is released, decreases as the job runs, and is set to 0 when the job completes. The function `defer` is used to calculate the slow-down factor by deferring work of currently released jobs. The detailed algorithm of `defer` [31] is listed in Algorithm 1, with a summary of relevant variables in Table 4.1.

Algorithm 1 Original `defer` function for LAEDF algorithm

Require: $n' \leq n$
Ensure: $0 < \alpha \leq 1$
  $U \Leftarrow \sum_{i=1}^{n'} \frac{C_i}{p_i}$
  $w \Leftarrow 0$
  for $i = 1$ to n' : $J_i \in \{J_1, J_2, \ldots J_{n'} | d_1 > d_2 > ... > d_{n'}\}$ **do**
    $U \Leftarrow U - \frac{C_i}{p_i}$
    $udw \Leftarrow \max(0, C\_left_i - (1 - U)(d_i - d_{n'}))$
    if $(d_i \neq d_{n'})$ **then**
      $U \Leftarrow U + \frac{C\_left_i - udw}{d_i - d_{n'}}$
    end if
    $w \Leftarrow w + udw$
  end for
  return $\alpha \Leftarrow \frac{w}{d_{n'} - t_{now}}$

Iterating from the job with the latest deadline J_1 (i.e., the lowest priority job) to $J_{n'}$, the job with the earliest deadline, `defer` computes how much work can be deferred after $d_{n'}$ for each job J_i. Some or all of J_i's work can be deferred until after

Table 4.1: Summary of key variables in the LAEDF algorithm

Variable	Explanation
n'	Number of tasks with a released job
U	Processor utilization required after d_n for higher priority tasks and deferred work
C_left_i	Remaining execution time of job J_i
udw	Amount of undeferable work for the current task
w	Total work that cannot be deferred after d_n by all tasks
α	Slowdown factor for current job J_i
t_{now}	The current time

$d_{n'}$ if work demanded by other jobs in the interval between $d_{n'}$ and d_i does not totally consume the processor on the interval $(d_{n'}, d_i]$. The amount of work that cannot be deferred, $udw = \max(0, C\_left_i - (1-U)(d_i - d_{n'}))$, where U is the amount of work demanded by higher priority jobs and deferred parts of lower priority jobs in $(d_{n'}, d_i]$.

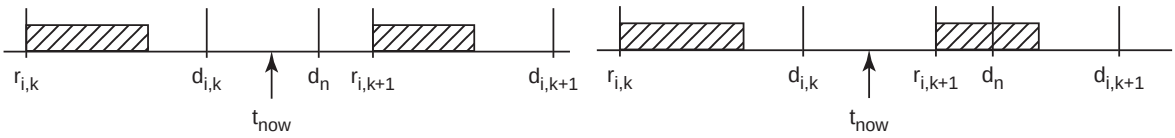
Initially, U is the system utilization, $\sum_{i=1}^{n'} \frac{C_i}{p_i}$. With each iteration, J_i's utilization is subtracted from U. If all the remaining work, C_left_i can be deferred, the amount of time it will demand before its deadline, $\frac{C\_left_i}{(d_i - d_{n'})}$, is added back to U. Otherwise, some part of J_i cannot be deferred, so the processor will be busy with J_i and higher priority jobs before d_i, and U is set to 1. The variable, w, accumulates the total undeferable work for all jobs. The slowest system speed required to finish the undeferable work before $d_{n'}$ is $\alpha = \frac{w}{(d_{n'} - t)}$. A slowest frequency available faster than α is selected.

At every job release or completion, `defer` is called to update the frequency based on the latest values of C_left_i.

4.4.2 Extension of LAEDF to Distributed LAEDF

As discussed in Section 4.1, when $D_i < p_i$, the utilization test for schedulability can be replaced with a density test to guarantee schedulability. However, when $D_i < p_i$, `defer` may be called, at time t_{now}, after a $J_{i,k}$ has passed its deadline, but before $J_{i,k+1}$ is released (i.e., $d_{i,k} < t_{now} < r_{i,k+1}$). In this case, which cannot occur when $D_i = p_i$, the undeferable work of $J_{i,k+1}$ is 0 when the next job will be released after the next deadline in the system (i.e., $r_{i,k+1} \geq d_{n'}$, as in Figure 4.1(a).)

However, If $r_{i,k+1} < d_{n'}$, as in Figure 4.1(b), $J_{i,k+1}$ may have some undeferable work. In this case, $d_{i,k+1}$ must be used to compute the amount of undeferable work, even though $J_{i,k+1}$ has not been released. Otherwise, too much work may be deferred until after $d_{n'}$ because no undeferable work from $J_{i,k+1}$ would be included when determining the minimum system speed. The value of C_left_i for this unreleased job



(a) $J_{i,k+1}$ has no deferable work before $d_{n'}$ (b) $J_{i,k+1}$ might have deferable work before $d_{n'}$

Figure 4.1: Cases not covered in original LAEDF

must be C_i because $J_{i,k+1}$ has not run. Algorithm 2 gives pseudocode for the modified `defer` function for DLAEDF algorithm.

Algorithm 2 Function `defer` for DLAEDF algorithm

Require: None
Ensure: $0 < \alpha \leq 1$
  $\Delta = \sum_{i=1}^{n'} \frac{C_i}{\min(p_i, D_i)}$
  $w \Leftarrow 0$
  for $i = 1$ to $n' : J_i \in \{J_1, J_2, \ldots J_{n'} | d_1 > d_2 > ... > d_{n'}\}$ **do**
    $\Delta \Leftarrow \Delta - \frac{C_i}{\min(D_i, p_i)}$
    if $(r_i > d_{n'})$ **then**
      $udw \Leftarrow 0$
    else
      if $(r_i > t_{now})$ **then**
        $C\_left_i \Leftarrow C_i$
      end if
      $udw \Leftarrow \max(0, C\_left_i - (1 - \Delta)(d_i - d_{n'}))$
    end if
    if $(d_i \neq d_{n'})$ **then**
      $\Delta \Leftarrow \Delta + \frac{C\_left_i - udw}{d_i - d_{n'}}$
    end if
    $w \Leftarrow w + udw$
  end for
  return $\alpha \Leftarrow \frac{w}{d_{n'} - t_{now}}$

LAEDF can also be extended for systems with deadlines longer than periods. When $D_i > p_i$, more than one job per task may be outstanding. To handle multiple jobs per task, `defer` should iterate through the set of jobs released before $d_{n'}$ regardless of to which task they belong, including any jobs to be released before $d_{n'}$ (as described for the $D_i < p_i$ case above.) No other changes are required to the algorithm if C_left_i is stored on a per job basis. The proper amount of time is reserved for all released jobs because C_left_i is set to C_i when each job is released.

4.4.3 Proof of Feasibility

Any set of tasks that meets the EDF schedulable utilization test is schedulable by LAEDF. More formally

Theorem 4.4.1. *A set of tasks is schedulable under DLAEDF if*

$$\Delta = \sum_{k=1}^{n} \frac{C_i}{\min(\mathrm{p_i}, \mathrm{D_i})} \leq 1$$

.

The proof strategy is to transform an EDF schedule into an DLAEDF schedule without rendering a feasible schedule infeasible.

Proof. A system scheduled with EDF is schedulable as long as the density, $\Delta \leq 1$ over every interval [22]. With LAEDF, jobs are scheduled in EDF order, but voltage and speed change at the scheduling points (job releases and completions.) As long as the speed changes do not cause density to exceed 1 on any interval, the system will remain schedulable.

At every scheduling point, `defer` considers each job in order of decreasing deadline ($d_1 > d_2 > \ldots > d_{n'}$). No work is deferred after d_1, the absolute deadline furthest in the future. The density on the interval (d_1, ∞) will simply be Δ, so the system will remain schedulable after d_1 as long as $\Delta \leq 1$.

Work from J_1 can be deferred after $d_{n'}$ without affecting feasibility as long as J_1 completes before d_1 and all other jobs complete before their deadlines. The density of jobs with higher priority than J_1 during the interval $(d_{n'}, d_1]$ is

$$\Delta_1 = \sum_{k=2}^{n'} \frac{e_k}{\min(p_k, D_k)}. \tag{4.5}$$

Up to $(1 - \Delta_1)(d_1 - d_{n'})$ units of work can be deferred past $d_{n'}$ without affecting feasibility. If $C\_left_i \leq (1 - \Delta_1)(d_1 - d_{n'})$ then all of the remaining work in J_i can be deferred. Otherwise the amount of work that cannot be deferred is

$$C\_left_i - (1 - \Delta_1)(d_1 - d_{n'})$$

.

Deferring work does not affect schedulability on $(t, d_{n'}]$ because work that would have been completed before $d_{n'}$ is moved after $d_{n'}$, reducing the work demanded of

the processor before $d_{n'}$. The interval $(d_{n'}, d_1]$ remains schedulable because density of all tasks scheduled on the interval is maintained at a value less than or equal to 1.

On the next iteration of the loop in `defer`, the density of jobs with higher priority than J_2 is

$$\Delta_2 = \Delta_1 + \min(\frac{C\_left_1}{d_1 - d_{n'}}, (1 - \Delta_1)) - \frac{C_2}{p_2}. \tag{4.6}$$

Assume the i^{th} job J_i and all preceding jobs remain schedulable after being transformed by LAEDF. The density of higher priority jobs for J_{i+1} is

$$\Delta_{i+1} = \Delta_i + \min(\frac{C\_left_i}{(d_i - d_{n'})}, (1 - \Delta_i)) - \frac{C_{i+1}}{p_{i+1}}, \tag{4.7}$$

and the amount of work deferred after $d_{n'}$ is

$$\min(C\_left_{i+1}, (1 - \Delta_{i+1})(d_i - d_{n'})). \tag{4.8}$$

As with J_1 the density of the work moved after $d_{n'}$ when added to Δ_{i+1} is less than or equal to 1, leaving the interval $(d_{n'}, d_{i+1}]$ schedulable. Deferring work on the interval $(t, d_{n'}]$ only reduces the density on the interval, thus deferring work from J_{i+1} after $d_{n'}$ does not affect schedulability of the system. By induction, deferring work from all jobs does not make the schedule infeasible.

After iterating over all jobs currently released, and those jobs that will be released before $d_{n'}$, the processor speed it set to the slowest speed fast enough to complete the total undeferable work from other jobs and C_left_n before $d_{n'}$. □

4.5 Distributed Feedback EDF

To extend the FEDF to distributed FEDF, the two challenges mentioned in the beginning of this chapter have to be solved. A description of the extension of FEDF is given in detail after the introduction of uniprocessor FEDF in the following section.

4.5.1 Uniprocessor Feedback EDF

Feedback EDF (FEDF) was originally proposed by Dudani, Mueller and Zhu [14], and has subsequently been refined by Zhu and Mueller [48, 49, 50]. Studies have shown that FEDF [14, 47, 48, 49] is able to reduce energy consumption more than LAEDF, though FEDF incurs more run-time overhead. As shown in Equation 2.8 and 2.9, FEDF gets its performance advantage from job splitting and slack estimation. Each

job is split into two parts based on the assumption that actual execution time is typically shorter than worst case execution time. The split is chosen so that the first part of each job is likely to be run, and the second part is not. The scheduler reserves enough time to run the unlikely part at full speed, creating more dynamic slack for the likely part of the job. Most of the time, the job completes before the unlikely part runs.

FEDF reserves time for jobs using a combination of slack passing from tasks that finish before their worst-case execution time and static slack in a precomputed maximal schedule. The static slack is distributed throughout the schedule by including an idle task, with period

$$p_{idle} \leq \min_{1 \leq i \leq n}(p_i). \tag{4.9}$$

This p_{idle} guarantees there will be at least one idle slot during the every period of each task in the maximal schedule. The WCET of the idle task is set such that the system utilization is one. The actual execution time of idle task is always zero.

At run time, FEDF keeps a track of slack time available for the current job to help reduce processor speed. The slack is gained from either idle slots in the maximal schedule and dynamic slack from jobs that finish early. The detailed algorithm for slack time tracking in FEDF is given in Algorithm 3 and Algorithm 4, with relevant variables in Table 4.2.

Algorithm 3 Calculation of slow-down factor in FEDF

Require: None
Ensure: $0 < \alpha \leq 1$
  if (J_{pk} is preempted) **then**
    if ($C\_left_{pk} > slots(J_{pk}, t_{now}, d_{pk})$) **then**
      $reserve_{pk} \Leftarrow C\_left_{pk} - slots(J_{pk}, t_{now}, d_{pk})$
      Reserve $reserve_{pk}$ in $idle(t_{now}, d_{pk})$
    end if
    $slack \Leftarrow slack - Max(idle(d_{ij}, d_{pk}), reserve_{pk})$
  else
    if ($t_{now} > d_{pk}$) **then**
      $slack \Leftarrow slack - idle(d_{pk}, t_{now})$
    end if
    $slack \Leftarrow slack + idle(d_{pk}, d_{ij})$
  end if
  return $\alpha \Leftarrow \frac{C_{ij}^A}{C_{ij}^A + slack}$

The slack time is updated at each scheduling point of task release. When a job,

Algorithm 4 Task completion in FEDF

Require: None

Ensure: $reserve_{ij} == 0$

Mark unused slots allocated for J_{ij} in maximal schedule as idle slots

Update estimation of C^A_{ij+1}

$C\_left_{ij} \Leftarrow C_i$

if $(reserve_{ij} > 0)$ **then**

Release reserved slots for J_{ij}

end if

Table 4.2: Key variables in FEDF algorithm

Variable	Explanation
$slack$	Current estimated system slack time
J_{ij}	Current job
J_{pk}	Previous job relative to J_{ij}
t_{now}	Current time
C_left_{pk}	Remaining execution time of J_{pk}
d_{ij}	Absolute deadline of J_{ij}
$reserve_{pk}$	Time slots have to be reserved for preempted job J_{pk}
C^A_{ij}	Estimated actual execution time of J_{ij}
α	Slowdown factor for current job J_{ij}
$idle(t_1, t_2)$	Idle slots from both idle task and completed jobs between $[t_1, t_2]$
$slots(J_{ij}, t_1, t_2)$	Amount of time allocated or reserved for J_{ij} between $[t_1, t_2]$
$Max(v_1, v_2)$	Returns maximum value between v_1 and v_2

J_{ij} in the ready queue is scheduled to run, a processor speed has to be selected based on the system's available slack time. The maximal schedule gives the information of slack time available within the time duration of $[t_{now}, d_{ij}]$. All the idle slots in the maximal schedule within $[t_{now}, d_{ij}]$ are contributed to the slack time for J_{ij}. However, if the deadline of previous job, J_{pk} is within $[t_{now}, d_{ij}]$, the old value of slack time has included idle slots from t_{now} to the deadline of J_{pk}, d_{pk}. In that case, only the idle slots within $[d_{pk}, d_{ij}]$ need to be added to the slack.

When d_{ij} is earlier than d_{pk}, the preemption may happen. If J_{pk} is preempted by J_{ij}, time slots have to be reserved for the preempted job to guarantee no deadline miss. J_{pk} may have been executed at a lower processor speed caused by DVS. It will takes longer to finish the rest worst case work than the time that have been allocated to J_{pk} in the maximal schedule. Some extra idle slots during $[t_{now}, d_{pk}]$ have to be reserved to J_{pk} until the total number of reserved time reaches the remaining worst case execution time of the preempted job, C_left_{pk}. The reservation can be done in two ways, forward sweep or backward sweep. Forward sweep reserves idle slots for J_{pk} from t_{now} toward d_{pk} up to C_left_{pk}. While the backward sweep does the reservation from d_{pk} backwards. Backwards sweep leaves more idle slots usable for current job by aggressively push the reservation of J_{pk} as late as possible to the job's deadline. The amount of reserved time is subtract from slack after the slots reservation.

After a job, J_{ij} is completed, there might be unused time that have been allocated for J_{ij}. Those slots are marked as idle slots which can be contributed to the slack for the next scheduled job. If J_{ij} was preempted, all the idle slots reserved for J_{ij} are released and marked as idle for the future use.

The actual execution time of the completed job J_{ij} is used to update the estimation of actual execution time for the next job instance in the same task. Different algorithms can be used in the execution time estimation. The simplest one is to use the average of actual executions of all previously completed jobs in the same task. Others such as weighted average and proportional-integral-derivative (PID) controller can also be used. The effectiveness of various estimation algorithms is highly dependent on the nature of the task's workload.

4.5.2 Extension of FEDF to Distributed FEDF

FEDF requires a few modifications to work when task relative deadlines are shorter than task periods [40]. When some tasks have relative deadlines shorter than their periods, setting idle task utilization to make system utilization 1 may cause missed

deadlines. For such systems, the idle task's utilization has to be set such that the system's *density* is one instead of its utilization. In addition, rather than setting the idle task period to the minimum of the task periods, it is set to the minimum of the task relative deadlines.

$$p_{idle} \leq \min_{1 \leq i \leq n}(min(p_i, D_i)) \tag{4.10}$$

Otherwise, the idle slot that occurs during job $J_{i,k}$ in task T_i may only occur between the deadline of $J_{i,k}$ and the release time of $J_{i,k+1}$. Using the minimum deadline guarantees at least one idle slot before each job's deadline.

Although the FEDF algorithm can work with systems having short deadlines, changes have to be made to handle the release time jitter at run time. FEDF calculates a maximal schedule offline using known information such as tasks' worst execution times and periods to help the algorithm exploit slack time at run time. The maximal schedule assumes the interrelease times of jobs are constant. Release jitter makes it impossible to use a static maximal schedule. To extend FEDF to handle release time jitter, a new maximal schedule has to be calculated each time a new job is released. The new algorithm for distributed FEDF (DFEDF) is given in Algorithm 5.

Algorithm 5 Task release of DFEDF

Require: $t_{now} < d_{ij}$
Ensure: $0 < \alpha \leq 1$
  if $(d_{ij} > d_{pk})$ **then**
    $slack \Leftarrow 0$
    Recompute maximal schedule from t_{now} till d_{ij}
    $slack \Leftarrow slack + idle(t_{now}, d_{ij})$
  else
    if (J_{pk} is preempted) **then**
      if $(C\_left_{pk} > slots(J_{pk}, t_{now}, d_{pk}))$ **then**
        $reserve_{pk} \Leftarrow C\_left_{pk} - slots(J_{pk}, t_{now}, d_{pk})$
        Reserve $reserve_{pk}$ in $idle(t_{now}, d_{pk})$
      end if
      $slack \Leftarrow slack - Max(idle(d_{ij}, d_{pk}), reserve_{pk})$
    else
      $slack \Leftarrow slack - idle(d_{ij}, d_{pk})$
    end if
  end if
  return $\alpha \Leftarrow \frac{C_{ij}^A}{C_{ij}^A + slack}$

When a job, J_{ij}, is scheduled to run by DFEDF, its absolute deadline, d_{ij}, has to be compared with the deadline of previously executed job, d_{pk}. If $d_{ij} > d_{pk}$, there

is no slot reservation information in the maximal schedule for the time duration of $[d_{pk}, dij]$, which means a new maximal schedule has to be recomputed before DFEDF can estimate slack time available for J_{ij}. To eliminate the error in maximal schedule that may caused by release jitter of current job, a new maximal scheduler during $[t_{now}, d_{ij}]$ is computed. The available slack time is reset to the idle slots within $[t_{now}, d_{ij}]$. If $d_{ij} \leq d_{pk}$, maximal schedule does not need to be updated. If a job is preempted, slots have to be reserved for the preempted job to ensure the timeliness. Otherwise the idle slots within $[d_{ij}, d_{pk}]$ is subtracted from the available slack for current job, J_{ij}.

The detail of maximal scheduler recomputation is given in Algorithm 6. The maximal schedule computation schedules the task set on the processor with EDF scheduling assuming the worst-case execution time for each job in the task. A linked list contains information of scheduling results. In each of the node, there are records of task name, starting time and endding time of the task's execution. The nodes (time slots) are linked according to the increasing order of starting time in the node. Besides the regular real-time tasks, an idle task is created and scheduled in the maximal schedule. The created idle task has its period equal to the minimum period among the task set and the worst-case execution time of

$$(1 - \Delta) \times p_{idle}.$$

Where Δ is the processor's total density. The time slots scheduled for idle task along with the time span between each busy interval in the maximal schedule are marked as idle slots, which contributes to the slack time for the job under scheduling in DFEDF.

4.5.3 Proof of Feasibility

Our proof extends a proof of correctness for Feedback EDF that assumes task relative deadlines equal to periods [50].

Theorem 4.5.1. *Any set of tasks with deadlines less than or equal to their periods that can be feasibly scheduled by EDF can also be feasibly schedulable with DFEDF.*

Proof. Feedback EDF produces feasible schedules for tasks with relative deadlines equal to their periods when system utilization does not exceed 1 (see [50] for a proof). Our strategy is to transform system $\mathbf{T}$ with relative deadlines less than or equal to periods into a system $\mathbf{T}'$ with deadlines equal to periods such that if $\mathbf{T}'$ is schedulable by Feedback EDF then $\mathbf{T}$ is schedulable by DFEDF.

Algorithm 6 Maximal schedule recomputation of DFEDF algorithm

Require: $t_{start} < t_{end}$
Ensure: None
  for (All subtasks on P_i) **do**
    Get jobs for subtasks
    $enqueue(job, waitingQ)$
  end for
  Add idle jobs into waitingQ
  $t \Leftarrow t_{start}$
  while ($t \leq t_{end}$) **do**
    for (All jobs in waitingQ) **do**
      if ($r_i \leq t$) **then**
        $dequeue(J_i, waitingQ)$
        $enqueue(J_i, readyQ)$
      end if
    end for
    Schedule jobs in readyQ using their WCETs at max processor speed
    Allocate time slots for scheduled jobs
    Advance t
  end while

Let $\mathbf{T}$ be a set of n tasks with each task T_i having period p_i, worst-case execution time C_i, and relative deadline D_i with $D_i \leq p_i$. Construct a new set, $\mathbf{T'}$, of n tasks such that for each $T_i \in \mathbf{T}$ there exists a $T'_i \in \mathbf{T'}$ such that $p'_i = \min(p_i, D_i)$, $C'_i = C_i$, and $D'_i = \min(p_i, D_i)$.

T' is schedulable by Feedback EDF when $U' = \sum_{k=1}^{n} C'_i/p'_i \leq 1$ by Zhu and Mueller's proof because each task's period equals its relative deadline [50]. A system scheduled by EDF remains schedulable if task interrelease times are actually longer than their periods [22], and Feedback EDF is capable of scheduling any set of tasks schedulable by EDF. Thus, $\mathbf{T'}$ remains schedulable by Feedback EDF even if the actual job interrelease times for task T'_i are p_i instead of $p'_i = \min(p_i, D_i)$, as long as $U' \leq 1$.

Since $U' = \sum_{k=1}^{n} C'_i/p'_i = \sum_{k=1}^{n} C_i/\min(p_i, D_i) = \Delta$, $\mathbf{T'}$ is schedulable by Feedback EDF when $\Delta \leq 1$. In the case where the actual interrelease times of each task in T'_i in $\mathbf{T'}$ are equal to the periods of each task T_i in $\mathbf{T}$, saying that T' is schedulable by Feedback EDF when $U' \leq 1$ is the same as saying $\mathbf{T}$ is schedulable by DFEDF when $\Delta \leq 1$, because DFEDF simply substitutes Δ for U, and handles the idle task in a slightly different way that does not affect schedulability. In either algorithm, idle time is computed based on a maximal schedule, and only time not used by any other task is allocated to idle slots. Therefore, $\mathbf{T}$ is schedulable by DFEDF when the

system's density does not exceed 1. □

4.6 Arbitrary Deadline CCEDF

The DCCEDF discussed earlier in this chapter uses the system density, instead of utilization, to determine the minimum execution speed for the current job running on a real-time system with tasks whose deadlines is different than their periods. The density test for such system, however, is not optimal. That is, the system's static slack time estimated using system's density is less than the actual value. To exploit more static slack time by using a tighter schedulability bound, the arbitrary dealdine CCEDF (ADCCEDF) is proposed. The detail of this algorithm is discussed in the rest of this section.

4.6.1 Schedulability Test with Tighter Bound

A real-time periodic task may has its relative deadline be equal to, greater than or less than its period. A system with such task is called a real-time system with arbitrary relative deadlines. Unlike the utilization test for system with task's deadline equal to its period, the optimal schedulability test for a real-time system with arbitrary relative deadlines scheduled by EDF has been studied and proved to be an NP-complete problem [17, 2]. The density test serves as an quick, however not quite accurate, schedulability test for the system with arbitrary deadlines. A new schedulability test has been shown to give a tighter bound than the density test for n tasks, and require $O(n)$ work when task information is available in order of non-decreasing relative deadline [12].

This new schedulability test is given by the equation below [12].

$$\hat{U}_k = \sum_{i=1}^{k} \frac{C_i}{p_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{p_i - \min(p_i, D_i)}{p_i} \right) C_i \leq 1. \tag{4.11}$$

Where $1 \leq k < n$. The system of n tasks with arbitrary deadlines is schedulable, if 4.11 is true for all tasks in the system.

4.6.2 Extension of CCEDF Using a Tighter Bound

To exploit dynamic slack in the system, rather than computing density based on worst-case execution time, C_i, ADCCEDF computes Equation 4.11 for each task using c_i in place of C_i, where c_i is set to the actual execution time at every completion and

the worst case execution time at every release, just as in CCEDF.

Unlike the computation of U in CCEDF, ADCCEDF must compute n values of $\hat{U}_k$, one for each task T_k. If the processor is scaled by some factor α, the scaled version of Equation 4.11 becomes

$$\frac{1}{\alpha}\sum_{i=1}^{k}\frac{c_i}{p_i}+\frac{1}{\alpha\cdot D_k}\sum_{i=1}^{k}\left(\frac{p_i-\min(p_i,D_i)}{p_i}\right)c_i\leq 1. \tag{4.12}$$

Solving for α we get:

$$\sum_{i=1}^{k}\frac{c_i}{p_i}+\frac{1}{D_k}\sum_{i=1}^{k}\left(\frac{p_i-\min(p_i,D_i)}{p_i}\right)c_i\leq \alpha. \tag{4.13}$$

Equation 4.13 must be true for all n tasks, thus

$$\alpha\geq \max_{1\leq k\leq n}(\hat{U}_k). \tag{4.14}$$

In other words the speed of the system can be set to lowest speed no less than the largest $\hat{U}_k$. Note that the test in Equation 4.11 is valid regardless of whether deadlines are shorter, longer, or equal to relative deadlines. As a result, ADCCEDF works with arbitrary deadlines.

CCEDF has very low computational complexity. At every job release and completion, the old contribution of the current task is subtracted from U, and the new contribution is added to it. Overall only $O(1)$ operations are required per task switch. In contrast, ADCCEDF must update all n of $\hat{U}_k$ values used in Equation 4.14. If the values of $\hat{U}_k$ are computed in order of non-decreasing relative deadlines, however, only $O(n)$ operations are required per task switch.

4.7 Summary

This chapter discussed the extensions of the applicability of uniprocessor DVS-EDF algorithms to tasks with end-to-end precedence constraints and deadlines different than their periods in partitioned distributed real-time system. The changes to SEDF and CCEDF are as simple as substituting density for utilization. ADCCEDF modifies CCEDF to use a tighter schedulability bound, allowing slower processor speeds for the same workload. DLAEDF requires more extensive changes. Its deferable work computation must account for jobs whose deadline has passed, but whose next job in the task has not been released. FEDF requires the largest change. To overcome the release jitter in the partitioned real-time systems, DFEDF has to compute the

available slack using a dynamically computed maximal schedule.

The simulations have been done to compare the energy-saving performance of the distributed DVS-EDF algorithms discussed in this chapter. The detailed simulation results are presented in Chapter 7.

Chapter 5: TASK ASSIGNMENT AND DEADLINE ASSIGNMENT

Task assignment and deadline assignment have to be done before scheduling the task set in distributed real-time systems using priority-driven scheduling algorithms. How dependent subtasks are assigned to the processors in the distributed system and how the deadline of each subtask within an end-to-end task is assigned will affect the schedulability of the system and the energy-conserving performance of DVS scheduling algorithms. This chapter discusses and compares several energy-aware online task assignment heuristics and deadline assignment approaches for distributed real-time system in terms of schedulability and energy conservation.

5.1 Task Assignment

To apply the priority-driven scheduling algorithms to distributed real-time system, partitioning-based scheduling is used.

Task assignment in partitioning-based scheduling can be done offline or online, as discussed in Chapter 2. To take the advantage of the priority-driven scheduling's ability to schedule dynamic real-time task sets, online task assignment heuristics with low online overhead are proposed [3, 45, 9].

Work by Adyin and Yang shows that how tasks are assigned to the processor effects the energy-conserving performance of various DVS algorithms [3]. However, looking for a feasible task assignment with minimized energy-consumption for a distributed real-time system has been proved an NP-Hard problem in the strong sense [3]. energy-aware task assignment heuristics with low online overhead are proposed to schedule dynamic task sets in distributed real-time systems.

We focus the discussion on the energy-aware task assignment heuristics for an EDF scheduled distributed real-time system. The problem of energy-aware task assignment for such a system is formally described as follows. Given the densities of set of subtasks in n end-to-end tasks, a partition of the task set onto m processors is desired such that the subtasks assigned on each processor are schedulable according to EDF scheduling algorithm. In order to simplify the discussion, the term *task*, instead of *subtask*, is used in the descriptions of task assignment algorithms as the assignment object. The performance of the assignment is evaluated by the energy consumption of the system using DVS-EDF scheduling algorithms. Several task assignment heuristics are discussed in the following subsection.

5.1.1 Simple Task Assignment Heuristics

There are a number of task assignment heuristics that can generate task assignments, such as First-Fit, Best-Fit and Next-Fit [15, 23, 8]. Unlike the offline task assignment algorithms discussed in Chapter 2, these heuristics assign tasks to the processors one by one. Among these algorithms, First-Fit and Worst-Fit are discussed in detail in the rest of this subsection.

To solve the task assignment problem we have formed previously, the Best-Fit assigns a subtask to the processor with minimum available computation capacity which can just fit the subtask as described in Algorithm 7 with the relevant variables in Table 5.1.

Algorithm 7 Best-Fit task assignment

Require: $U_j = 0, 0 < j \leq m$
Ensure: $U_j \leq 1, 0 < j \leq m$
  for (Each task $T_i \in \boldsymbol{T}$) **do**
    $\delta_i \Leftarrow C_i / min(p_i, D_i)$
    $U_{best} \Leftarrow max_{0<j\leq m}(U_j)$
    while ($\delta_i < 1 - U_{best}$) **do**
      if ($U_{best} > min_{0<j\leq m}(U_j)$) **then**
        $U_{best} \Leftarrow$ greatest $U_k < U_{best}$
      else
        No feasible assignment, Exit
      end if
    end while
    Assign T_i onto P_{best}
    $U_{best} \Leftarrow U_{best} + \delta_i$
  end for

The Best-Fit makes assignment decisions based on the task's density. The processor with the minimum remaining computation capacity, evaluated by $1 - U_{best}$, that can just accommodate the task, $U_{best} + \delta_i \leq 1$, is selected by the Best-Fit as the target for the current task, T_i. The Best-Fit algorithm tries to assign as many task to the same processor as possible, which leading to a fewer number of processors used by the system.

The Worst-Fit algorithm, on the contrary, assigns tasks onto the processor with maximum computation capacity available. Algorithm 8 gives the pseudo code for the Worst-Fit assignment.

A Worst-Fit assignment tends to produce a balanced workload. That is, the system's static slack time is distributed evenly among processors. The balanced

Table 5.1: Key variables used in task assignment algorithms

Variable	Description
P_i	The i^{th} processor in the system
$\boldsymbol{P}$	The set of all processors in the system
C_i	Execution time of task T_i
D_i	Relative deadline of task T_i
p_i	Period of task T_i
U_j	Total utilization of processor P_j
Δ_j	Total density of j^{th} processor
u_i	Density of task T_i
$\boldsymbol{T}$	The set of tasks to be assigned to processors
δ_i	Utilization of task T_i
P_{min}	Processor with minimum increase in average power
ΔPwr_{min}	Estimated increase in average power for P_{min}
Pwr_{idle}	Idle power consumption of processor
power(x)	Power consumption according to CPU speed x
pred(T_i)	Task T_i's predecessor
proc(T_j)	Processor to which task T_j is assigned
msg_energy(T_i)	Energy consumed by communication between T_i and its predecessor

Algorithm 8 Worst-Fit task assignment

Require: $U_j = 0, 0 < j \leq m$
Ensure: $U_j \leq 1, 0 < j \leq m$
  for (Each task $T_i \in \boldsymbol{T}$) **do**
    $\delta_i \Leftarrow C_i/min(p_i, D_i)$
    $U_{worst} \Leftarrow min_{0<j\leq m}(U_j)$
    if $(\delta_i le(1 - U_{worst}))$ **then**
      Assign T_i onto P_{worst}
      $U_{worst} \Leftarrow U_{worst} + \delta_i$
    else
      No feasible assignment, Exit
    end if
  end for

workload distribution enables all of the processors in the system to run as slow as possible. On the other hand, the Best-Fit task assignment results in an unbalanced workload distribution among processors. When static power is small, distributing the load evenly minimizes overall energy consumption because each processor is running at its minimum speed. When static power is a significant contribution to overall power, it may be more advantageous to turn off some processors completely and run the remaining processors at a higher speed.

5.1.2 Communication-Aware Worst-Fit

Subtasks within an end-to-end task communicate to maintain the dependencies. The cost of communication increases when two adjacent subtasks are assigned onto different processors. Neither the Best-Fit nor the Worst-Fit heuristic discussed previously accounts for communication between subtasks. The Worst-Fit heuristic is good at balancing the system's static slack among processors which can be used by DVS-EDF scheduling algorithms. The balancing, however, may increase the communication among processors. In order to take the cost of communication among subtasks, a communication-aware Worst-Fit (CAWF) is proposed.

Best Fit and Worst Fit both ignore communication costs between tasks. Synchronization signals and data have to be transmitted through the network when two dependent tasks are assigned to different processors. To account for the communication cost caused by signals and data transmission, we introduce Communication-Aware Worst Fit (CAWF) task assignment to reduce system communication cost while distributing the workload evenly among the processors. Algorithm 9 shows the pseudo code for CAWF task assignment algorithm.

When assigning a task to a processor, the CAWF algorithm checks which processor the task's predecessor is assigned to and tries to assign the task onto the same processor. When the density on the predecessor's processor becomes too high to accommodate the new task, the Worst Fit processor is selected for the next task. The Worst Fit algorithm is applied when the task has no predecessor.

Assigning tasks to the same processor reduces communication cost because the tasks do not need to send synchronization or data messages over the network. Selecting the Worst Fit processor in CAWF not only balances the load, as in Worst Fit, but also it leaves the maximum available density for subsequent tasks that depend on the task currently being assigned.

Algorithm 9 CAWF task assignment

Ensure: $\Delta_j \leq 1, 0 < j \leq m$
  for (Each task $T_i \in \boldsymbol{T}$) **do**
    $\delta_i \Leftarrow C_i/min(p_i, D_i)$
    $P_k \Leftarrow \text{proc}(\text{pred}(T_i))$
    if $(P_k \neq nil \wedge \delta_i \leq 1 - \Delta_k)$ **then**
      Assign T_i on processor P_k
      $\Delta_k \Leftarrow \Delta_k + \delta_i$
    else
      $\Delta_{min} \Leftarrow \min_{0<j\leq m}(\Delta_j)$
      if $(\delta_i \leq 1 - \Delta_{min})$ **then**
        Assign T_i on processor P_{min}
        $\Delta_{min} \Leftarrow \Delta_{min} + \delta_i$
      else
        No feasible assignment
      end if
    end if
  end for

5.1.3 MinΔP Task Assignment

Though they have been used for power-aware scheduling, Best-Fit and Worst-Fit do not assign tasks based on their power consumption. Algorithm 10 introduces a new greedy task assignment algorithm, called MinΔP. Recall that we have discussed that the voltage scalable processor can operate under a set of different voltage levels. The processor's speed and power consumption vary as the supply voltage changes. The set of voltage levels the processor can work with, however, is discrete. When there is a workload adding to the processor, in order to catch the tasks' deadlines, processor may have to run at a faster speed with a higher voltage level, thus a higher power consumption. If the new workload is not heavy enough requiring a processor speedup, the processor can accept this task without consuming more power. The MinΔP is proposed based on the above discussion. This algorithm assigns a task to the processor which will result in the smallest estimated increase in average power consumption on that processor.

The average power consumption of processor is used frequently in the MinΔP algorithm. The algorithm estimates the average power consumption of the processor in two parts, the processor execution power and the idle power. The execution power is estimated using the `power` function, which takes a processor frequency as an input and returns the power consumed while running at that frequency. The frequency of the processor is determined by the total density of the processor, but the processor

Algorithm 10 MinΔP task assignment

Require: $\Delta_j = 0, 0 < j \leq m$
Ensure: $\Delta_j \leq 1, 0 < j \leq m$
  for (Each task $T_i \in \boldsymbol{T}$) **do**
    $\delta_i \Leftarrow C_i / min(p_i, D_i)$
    $u_i \Leftarrow C_i / p_i$
    for (Each processor $P_j \in \boldsymbol{P}$) **do**
      if $(\delta_i + \Delta_j \leq 1)$ **then**
        if $(U_j > 0)$ **then**
          $\Delta Pwr \Leftarrow \text{power}(\delta_i + \Delta_j) \cdot (u_i + U_j) - \text{power}(\Delta_j) \cdot U_j - Pwr_{idle} \cdot u_i$
        else
          $\Delta Pwr \Leftarrow \text{power}(\delta_i) u_i + Pwr_{idle} \cdot (1 - u_i)$
        end if
        if $(\text{pred}(T_i) \neq nil$ and $\text{proc}(\text{pred}(T_i)) \neq P_j)$ **then**
          $\Delta Pwr \Leftarrow \Delta Pwr + \text{msg\_energy}(T_i)/p_i$
        end if
        if $(\Delta Pwr < \Delta Pwr_{min})$ **then**
          $\Delta Pwr_{min} = \Delta Pwr$
          $P_{min} \Leftarrow P_j$
        end if
      end if
    end for
    Assign task T_i to processor P_{min}
    $\Delta_{min} \Leftarrow \Delta_{min} + \delta_i$
  end for

is not always busy even if $\Delta = 1$. The processor's utilization better indicates how busy the processor will be. The power consumed when the processor is not busy is considered idle power. The average power consumption of the processor running at frequency of f can thus be estimated using the sum of execution power of

$$power(f) \cdot U$$

and the idle power of

$$Pwr_{idle} \cdot (1 - U)$$

$$Pwr = power(f) \cdot U + Pwr_{idle} \cdot (1 - U) \tag{5.1}$$

Where U is the processor utilization. Pwr_{idle} is the power when the processor is idle.

For each task to be assigned, MinΔP computes the task's density, δ_i, and utilization, u_i. The density of the task is used to compute the required processor speed. The utilization is used as the workload adding to the processor. It's obvious that the task can only be assigned to those processors with enough computation capacity. That is, $\Delta_j + \delta_i \leq 1$. To make the decision on the assignment, the estimated increase in average power of each processor with enough computation capacity, ΔPwr, is computed according to Equation 5.1.

$$\begin{aligned} \Delta Pwr &= \text{power}(\delta_i + \Delta_j) \cdot (u_i + U_j) - \text{power}(\Delta_j) \cdot U_j - Pwr_{idle} \cdot (1 - U_j - (1 - U_j - u_i)) \\ &= \text{power}(\delta_i + \Delta_j) \cdot (u_i + U_j) - \text{power}(\Delta_j) \cdot U_j - Pwr_{idle} \cdot u_i \end{aligned} \tag{5.2}$$

If the processor has no tasks assigned to it, we assume the processor is off and consumes no power. Thus the ΔPwr in this case is

$$\Delta Pwr = \text{power}(\delta_i) \cdot u_i + Pwr_{idle} \cdot (1 - u_i) \tag{5.3}$$

The ΔPwr computation assumes that the processor's frequency is proportional to the task density on the processor. For SEDF, this is precisely correct. For other algorithms, the actual average power is lower, but this approximation works well in practice.

This algorithm takes care of communication power consumption. Besides the

increase of processor power consumption when assigning a task onto a processor, there might be additional communication power consumption if the task is assigned to the different processor from its predecessor. The processor on which the task's predecessor was assigned is located by functions `pred(`T`)` and `proc(`T`)`. Function `pred(`T`)` looks for the predecessor of task T, while `proc(`T`)` locates the processor the task was assigned to. If these two tasks are assigned on different processors, power consumption of `msg_energy(`T_i`)`$/p_i$ is added to ΔPwr.

The processor with minimum power increase after adding the task is selected as the target for that task. In MinΔP, each task is treated separately to increase speed and simplicity. An extension to the algorithm that tries several combinations of tasks may produce task assignments with lower average power than Min ΔP, but the overhead would be too high to be used as an online task assignment algorithm.

5.1.4 Summary

How tasks are assigned to processors in partitioned multiprocessor systems affects the energy-conserving performance of DVS-EDF scheduling algorithms. Besides the two existing heuristics, Worst-Fit and Best-Fit, two new task assignment algorithms, Communication-Aware Worst-Fit and MinΔP, are proposed. The Communication-Aware Worst-Fit tries to assigns tasks that communicating on the same processor to reduce overall communication cost. Tasks do not communicate or cannot be placed with their sibling tasks are assigned using Worst-Fit to balance the system workload. MinΔP uses estimated change in power to decide to which processor a task should be assigned.

Among the four task assignment algorithms, the Best-Fit and Worst-Fit need the least information about the task set when making task assignment decisions – just the current processor and task densities. Communication-Aware Worst-Fit additionally needs to know each task's predecessor as well as the predecessor's processor. MinΔP requires not only the task information, but also power cost for processors and communication. The simulations are conducted in Chapter 7 to reveal the energy-conserving performance of each task assignment discussed above.

5.2 Deadline Assignment

As we have briefly discussed in Chapter 2, deadline assignment is used to assign local deadline to each of the subtasks in the end-to-end task. For a priority-driven scheduling algorithm, the information of task's priority has to be known in order

to generate a feasible schedule. Assigning local deadlines to subtasks is equal to assigning the priorities of tasks that the scheduler works on. The problem of how to assign priorities to subtasks such that a priority-driven scheduler can generate a feasible schedule for all the subtasks on the processor has been proved to be NP-hard [22]. In order to deal with the dynamic task set that has tasks arriving or leaving the system when the system is running, a simple heuristic has to be used. We focus on the discussion of deadline assignment heuristics with low online overhead.

Deadline assignment may affect the performance of DVS scheduling algorithm just as task assignment does. The static slack is assigned to each of the subtask according to the deadline assignment. There are several existing deadline assignment heuristics, which will be discussed in the following subsection. Another deadline heuristic is proposed following the discussion of existing ones. The new heuristic is trying to facilitate the energy saving for DVS-EDF scheduling algorithms.

5.2.1 Existing Deadline Assignment Heuristics

There are four existing deadline assignment heuristics, ultimate deadline (UD), effective deadline (ED), proportional deadline (PD), and normalized proportional deadline (NPD) [22, 38]. They are defined as follows. For the k^{th} subtask, $T_{i,k}$, in an end-to-end task, T_i,

$$UD_{i,k} = D_i \tag{5.4}$$

$$ED_{i,k} = D_i - \sum_{l=k+1}^{n(i)} C_{i,l} \tag{5.5}$$

$$PD_{i,k} = D_i \frac{C_{i,k}}{C_i} \tag{5.6}$$

$$NPD_{i,k} = D_i \frac{C_{i,k}U(V_{i,k})}{\sum_{l=1}^{n(i)} C_{i,l}U(V_{i,l})} \tag{5.7}$$

Where D_i and C_i is the end-to-end deadline and worst-case execution time of the end-to-end task T_i, respectively. And $C_{i,k}$ is the worst-case execution time of the k^{th} subtask of T_i. The number of subtasks in task T_i is represented by $n(i)$. $U(V_{i,k})$ is the total utilization of processor with subtask $T_{i,k}$ running on it. And $V_{i,k}$ is the name of the processor on which the subtask $T_{i,k}$ executes.

The four deadline assignment heuristics are illustrated by an example given in

Table 5.2: An example of deadline assignment

$T_{i,k}$	$V_{i,k}$	$D_i = p_i$	$C_{i,k}$	$\bar{C}_{i,k}$	$UD_{i,k}$	$ED_{i,k}$	$PD_{i,k}$	$NPD_{i,k}$	$ANPD_{i,k}$
$T_{1,1}$	P_1	50	2	1	50	45	22.22	11.91	21.98
$T_{1,2}$	P_2	50	1	0.7	50	44	11.11	2.34	6.04
$T_{1,3}$	P_1	50	6	1	50	50	66.67	35.75	21.98
$T_{2,1}$	P_1	10	3	2	10	9	2.5	8.84	9.11
$T_{2,2}$	P_2	10	1	0.5	10	10	7.5	1.16	0.89
$T_{3,1}$	P_2	30	3	1	30	27	15	8.46	4.07
$T_{3,2}$	P_1	30	3	2.5	30	30	15	21.54	25.93

Table 5.2. There are three end-to-end tasks and two processors in the system, T_1, T_2, and T_3, with 3, 2 and 2 subtasks within them, respectively. The end-to-end deadline of each task is equal to its period given in column 3 in Table 5.2. The example assumes that subtasks have been assigned using a task assignment. Subtasks $T_{1,1}$, $T_{1,3}$, $T_{2,1}$, and $T_{3,2}$ are assigned on processor P_1. While $T_{1,2}$, $T_{2,2}$, and $T_{3,1}$ are assigned on P_2. The worst-case execution time, $C_{i,k}$, and average execution time of each subtask, $\bar{C}_{i,k}$, is given by column 4 and 5, respectively. The total utilization of P_1 and P_2 are 0.56 and 0.22, respectively.

UD sets deadlines of all the subtasks to their end-to-end deadline as illustrated in Table 5.2. This approach needs no other information about the task except its end-to-end deadline. However, UD ignores the fact that the subtasks shares the deadline with a total amount no greater than the end-to-end deadline, which may cause missed deadlines. The ED assignment assigns the local deadline to a subtask with a value of the end-to-end deadline minus the total worst-case execution time of its successors. This approach takes the execution of successor subtasks into account. Thus ED is expected to result in a better system schedulability.

The proportional deadline assignment allocates deadlines proportional to the subtask's worst-case execution time. Static slack of the end-to-end task is also assigned to each of the subtasks proportional to the subtask's worst-case execution time. The longer the worst-case execution time, the longer relative deadline and thus more slack time is assigned. In comparison with UD and ED, PD is the best in terms of schedulability. For DVS-EDF scheduling algorithms, PD facilitates their performance by assigning the static slack in a balanced way.

When the workload on each of the processors in the system is same, PD has the same performance with NPD. However, for the unbalance workload in the example above, the NPD is expected to be better in distributing the available slack by evenly assigning the deadline to the subtasks according to the processor's workloads. For a

heavily loaded processor, the subtasks assigned on that processor are assigned with longer deadlines, thus more slack time by NPD. The end-to-end task T_3 in the example has two subtasks with identical worst-case execution time of 3, which results in a identical PD for each of the subtasks. The NPDs, however, are different. This is because that these two subtasks are assigned to two different processors with different utilizations. $T_{3,1}$ is assigned to P_2, which has a lighter workload with utilization of 0.22 than that on P_1. A longer NPD is assigned to $T_{3,2}$ on P_1 to allocate more slack on P_1. PD is trying to evenly distributes the slack in the end-to-end task to each of the subtasks within that end-to-end task. While NPD means to balancing the system's static slack among the subtasks in all the end-to-end tasks.

When assigning deadline with PD and NPD, the system is more likely to be schedulable. However, more information is needed for PD and NPD. NPD needs the information of workload on each of the processors as well as the information of worst-case execution time of each subtasks. When applying the deadline assignment online, the trade-off between system schedulability and online overhead has to be considered.

5.2.2 NPD Using Average Execution Time

In the real world, the actual execution time of a real-time task is usually less than its worst-case execution time. Instead of the worst-case execution time, the NPD can use the subtask's estimated actual execution time when making deadline assignment. The actual execution time is randomly distributed between the minimum and the worst-case execution time. The average of a series of random numbers is usually used as the estimation of next random number. A new deadline assignment, Average-execution-time NPD (ANPD), is proposed to distribute the system's slack more efficiently by using subtask's average execution time. This assignment heuristic is defined in Equation 5.8.

For the k^{th} subtask, $T_{i,k}$, in an end-to-end task, T_i,

$$ANPD_{i,k} = D_i \frac{\bar{C}_{i,k} U(V_{i,k})}{\sum_{l=1}^{n(i)} \bar{C}_{i,l} U(V_{i,l})} \tag{5.8}$$

Where $\bar{C}_{i,k}$ is the average execution time of subtask $T_{i,k}$. The ANPD for each subtask in previous example is given in column 10 of Table 5.2. Subtasks $T_{1,1}$ and $T_{1,3}$ are assigned on the same processor P_1. Although they have different WCET of 2 and 6, respectively, their average execution times are same. The identical average execution times result in the same ANPDs for subtasks $T_{1,1}$ and $T_{1,3}$. For subtasks in T_2, an even smaller ANPD is assigned to $T_{2,2}$ than NPD. A very small deadline

assignment happens because that the ratio of the difference between the average execution times of two subtasks in T_2 to their total average execution time is greater than that between the worst-case execution time to their total worst-case execution time.

In the previous example, we noticed that the deadline assigned to subtask $T_{2,2}$ is 0.89, the value of which is less than $T_{2,2}$'s WCET of 1. Although 0.89 is still greater than $T_{2,2}$'s average execution time, it is possible that $T_{2,2}$ misses its local deadline when its actual execution time is greater than 0.89, the value of local deadline. The same situation may happens to NPD, too. It is because that NPD and ANPD balance the workload among processors by assigning shorter local deadlines to subtasks on the lightly loaded processors and longer deadlines to those on the heavily loaded ones. It is possible for a subtask on a lightly loaded processor be assigned with a local deadline very close to or even shorter than its WCET. The local deadline shorter than the subtask's WCET may cause deadline misses when scheduling with EDF-based scheduling algorithms.

5.2.3 Summary

Among all the deadline assignment heuristics discussed above, PD and NPD are expected to result in a higher schedulability than all other deadline assignment approaches [38, 22]. ANPD is proposed to take advantage of shorter actual execution time than task's WCET to saving more energy consumption. When applied to the distributed real-time system with tasks frequently arriving or leaving the system, the overhead of each deadline assignment has to be considered. NPD and ANPD need global information of utilization on each processor, which require higher online overhead than PD. In addition, it is observed in an example that ANPD assigns a local deadline to the subtask with a value less than the task's WCET. This means that the subtask may miss its deadline when its actual execution time is greater than the deadline that have been assigned to it. The further study on NPD shows that the same situation may happen to NPD also. Although it is feasible as long as the end-to-end deadline is guaranteed [38], the very short local deadlines might degrade the energy-saving performance of DVS-EDF scheduling algorithms.

Simulations are done to compare the effect of each deadline assignment on the energy-saving performance of DVS-EDF algorithms. The detailed discussion of simulation results is given later in Chapter 7.

Chapter 6: DYNAMIC TASK SET ADMISSION TEST

Many embedded systems must operate under relatively harsh conditions with meager resources. They have dynamic load, either due to events in the environment, variations in workload, or adding and removing tasks from the system. For example, in a target tracking application, a new task may be introduced to maintain the track as the tracked object moves. Interfacing with the real world implies timing constraints for sampling sensors and/or controlling actuators. Thus, the system needs a real-time scheduling algorithm capable of guaranteeing all tasks meet their deadlines and managing power consumption to extend battery life as much as possible.

As discussed in Chapter 2, static real-time schedulers repeatedly execute a statically computed schedule that is pre-computed offline. Computing the schedule offline has the benefit that complex algorithms can be used to find optimal solutions [34]. However, static schedules do not handle dynamically changing task sets well. Adding new tasks is difficult because the entire schedule must be recomputed, which often requires an expensive offline algorithm. Priority-driven scheduling algorithms are better suited to applications with these dynamic properties. An arriving periodic task may be scheduled immediately if it passes a simple admission test by priority-driven scheduled systems.

In a DVS scheduled real-time system, a DVS scheduler stretches process execution times to take advantage of slack in the system. When a new periodic task joins the system, this stretching may have deferred enough work to cause a job in the arriving task or a job in a currently admitted task to miss its deadline, even if the arriving task could have been feasibly scheduled under EDF. That is, a schedulability test that is sufficient to test for EDF schedulability is *not* sufficient for admitting new tasks when using DVS algorithms based on EDF.

This chapter introduces two algorithms for admitting tasks to systems scheduled by CCEDF, LAEDF, Feedback EDF, or other EDF-based DVS scheduling algorithms. The Generalized Admission Test determines if a new task can be admitted immediately without causing a deadline to be missed. The Feasible Deadline Computation algorithm is useful if the relative deadline of the first job in the newly arriving periodic task can be longer than for subsequent jobs in the task. It computes a deadline for the first job in the arriving periodic task that will guarantee that all tasks will remain schedulable.

6.1 Background

Real-time DVS algorithms that compute a schedule offline cannot handle adding new tasks well. Though their schedules approach the minimum possible expected energy consumption, computing the schedule is too expensive to do online. Often mixed integer linear programming, or algorithms with similar complexity are used [34].

Adding tasks in a priority-driven scheduler allows tasks to be added with much less computational complexity. The scheduler always runs the task with the highest priority that is available to run. When a new task arrives it is scheduled with all of the existing tasks after running a simple admission test decides if the new task can be scheduled. Typically a utilization test is used to determine if a new periodic task can be admitted. To our knowledge no existing admission tests are designed to work for DVS scheduling algorithms. The standard utilization tests will not work because they do not account for work that has been deferred to slow down the processor.

Sporadic job scheduling uses an admission test on individual sporadic jobs as they arrive, admitting those jobs that pass the test [22]. DVS algorithms and admission tests have been devised for these sporadic jobs. Sporadic job scheduling is not appropriate for hard real-time systems when the incoming task is a periodic task because the admission test will have to be run on each job in the task. Each job in the periodic task will have to be admitted to the system individually, with the possibility that some jobs may be rejected. Our tests allow the entire task to be admitted or rejected. That is, once the periodic task is admitted, all of its jobs are guaranteed to be able to run and complete by their deadlines. With sporadic job scheduling each job has to be tested for admission and may fail to be admitted.

A system is composed of n independent, preemptable periodic tasks. Each *periodic task*, T_i is a sequence of jobs, $J_{i,1}, J_{i,2}, ...J_{i,n}$ with worst case execution time C_i, period p_i, and relative deadline D_i. The jobs within task T_i are first eligible to execute at their *release times*, $r_{i,1}, r_{i,2}, ..., r_{i,n}$, and must complete by their *absolute deadlines*, $d_{i,1}, d_{i,2}, ..., d_{i,n}$. The inter-release time between jobs in T_i is at least p_i, but may be longer.

When a new task T_{new} arrives in a system without DVS, an admission test determines if scheduling T_{new} with the existing tasks will result in a feasible schedule. The new task can be admitted if the density $\delta_{new} + \Delta \leq 1$ and tasks are scheduled with EDF.

Though we are concerned with periodic tasks, it is often helpful to analyze the current, partially completed job in a task as a sporadic job. Like a jobs in a periodic

task, a *sporadic job* $J_{s,i}$ has a release time $r_{s,i}$, a worst case execution time $C_{s,i}$, and an deadline $d_{s,i}$. Unlike a job in a periodic task, a sporadic job is not released at regular intervals.

Instantaneous utilization can be used to decide whether a set of sporadic jobs are schedulable. The *instantaneous utilization* of a set of sporadic jobs is defined as

$$\hat{U} = \sum_{T_i \in T_{active}} \frac{C_i}{d_i - r_i},$$

where T_{active} is the set of all jobs that have been released, but have not yet reached their deadlines. A set of sporadic jobs is schedulable by EDF if $\hat{U} \leq 1$ at all times [11]. Moreover, a combination of periodic jobs with density Δ and sporadic jobs with instantaneous utilization $\hat{U}$ is schedulable if

$$\Delta + \hat{U} \leq 1 \tag{6.1}$$

at all times [11].

DVS algorithms slow down the processor to reduce power consumption when slack is available. For example, LAEDF defers as much work as possible after the next deadline in the system [31]. Very often jobs take less than the worst case execution time. As a result much of the time allocated for defered work is never used. This dynamic slack can be used to execute other jobs at slower speed.

The admission test given in Equation 6.1 assumes that new tasks can use static slack that DVS scheduling algorithms use to slow down the processor. If T_{new} is admitted, existing jobs may have consumed all the available slack by running at lower speeds. Even if Equation 6.1 is satisfied, the admission test must verify that enough slack is available before releasing the new task. If not enough slack is available to start a task that passes the test in Equation 6.1 immediately, it is possible to schedule the task if the application can allow the first job's deadline to be extended.

6.2 Adding Tasks with DVS

DVS algorithms can exploit slack available in the schedule to aggressively defer work. In addition to using time that would have been available in the worst case schedule, deferring work takes advantage of slack available from jobs that take less than their worst case execution time. However, deferring work reduces the probability that an arriving task will be schedulable, because less slack is available in the schedule — the more effective the DVS algorithm at using slack, the less likely the new task can be

scheduled immediately.

6.2.1 Generalized Admission Test

Priority-driven DVS algorithms based on EDF are designed to schedule any task set schedulable by EDF [31, 50]. Proofs of schedulability show that as long as a set tasks can be scheduled by EDF, they can be scheduled by the DVS algorithm. For such algorithms, it is sufficient to show that an arriving task T_{new} can be scheduled by EDF with the existing tasks, including any work deferred by the DVS algorithm.

As long as $\Delta + \hat{U} \leq 1$ holds, any set of preemptable sporadic jobs and periodic tasks are schedulable by EDF [22]. Let t be the time at which a new periodic task arrives and the admission test is run. At time t, the current job $J_{i,k}$ in every task T_i can be treated as a sporadic job with an execution time of $C_{left,i}$ until its deadline $d_{i,k}$, where $C_{left,i}$ is the remaining amount of execution time for the current job in T_i. After $d_{i,k}$, task T_i is treated as a periodic task. Tasks with no active jobs are always treated as periodic tasks.

The deadlines of the sporadic jobs divide time into intervals. Before each deadline, the instantaneous utilization is calculated using the sporadic job parameters, but after the deadline, instantaneous utilization is calculated using the associated periodic task. In the worst case, the deadlines of all sporadic jobs must be tested.

In a system of n tasks, let $T_1, T_2, ..., T_m$ be tasks, including T_{new}, whose current deadline has not passed that are ordered such that their deadlines are $d_1 \leq d_2 \leq ... \leq d_m$. Let $T_{m+1}, T_{m+2}, ..., T_{n+1}$ be tasks whose current deadline has passed, but for which the next job has not been released[1]

Theorem 6.2.1. *A DVS scheduling algorithm that can feasibly schedule any set of n tasks schedulable by EDF can also feasibly schedule T_{new} with the n existing tasks at time t if*

$$\Delta_j = \sum_{i=j}^{m} \frac{C_{left,i}}{d_i - t} + \sum_{i=1}^{j-1} \frac{C_i}{\min(p_i, D_i)} + \sum_{i=m+1}^{n+1} \frac{C_i}{\min(p_i, D_i)} \leq 1, \forall j : 1 \leq j \leq m+1, \quad (6.2)$$

Proof. A job that has been released from a periodic task can be analyzed as a sporadic job with execution time of its remaining worst case execution time, $C_{left,i}$, deadline of its absolute deadline, d_i, and release time of current time t. The periodic tasks

[1]There may be a delay between the completion of one job in a task and the release of the next job either because the task has a relative deadlines shorter than its period or the next job release is waiting for an event.

without released jobs are treated as normal periodic tasks.

The new task, T_{new} can be divided into a sporadic job of its first released job $J_{new,1}$ and a periodic task with the rest of periodically released jobs. If the application allows the deadline of the first job in T_{new} to be longer than subsequent deadlines, then for intervals before $d_{new,1}$ we compute the instantaneous density of $J_{new,1}$ as $\delta_{new,1} = C_{new}/(d_{new,1} - r_{new,1})$. Otherwise the new task's density is $\delta_{new} = C_{new}/\min(p_{new}, D_{new})$.

Time can be separated into intervals by the deadlines of each sporadic job. Since a sporadic job is active only in its feasible interval $[r_i, d_i]$, the active sporadic jobs in interval I_k are $J_i, i = k, k+1, ..., m$.

It has been shown that a system of independent, preemptable sporadic jobs is schedulable by EDF if the total density of all active jobs in the system is no greater than 1 at all times [22]. Based on this theorem, the system is schedulable by EDF if

$$\Delta_{s,k} + \Delta \leq 1 \tag{6.3}$$

for all time, where $\Delta_{s,k}$ is the total density of active sporadic jobs in interval $[d_{k-1}, d_k]$, Δ is the total density of periodic tasks.

For the k_{th} interval I_k, density due to jobs with uncompleted work in the current job is

$$\Delta_{s,k} = \sum_{i=k}^{m} \frac{C_{left,i}}{d_i - t}, \tag{6.4}$$

including the first job in T_{new} if its deadline is allowed to be longer than the relative deadline of other jobs in T_{new}. The density due to periodic tasks with no active jobs, but that may release a new job at any time is

$$\Delta = \sum_{i=1}^{k-1} \frac{C_i}{min(p_i, D_i)} + \sum_{i=m+1}^{n+1} \frac{C_i}{min(p_i, D_i)}, \tag{6.5}$$

including T_{new} in the first sum if $d_{new,1} \neq D_{new}$ or in the second sum otherwise.

By substituting $\Delta_{s,k}$ and Δ in (6.3) with (6.4) and (6.5), we get

$$\Delta_k = \sum_{i=k}^{m} \frac{C_{left,i}}{d_i - t} + \sum_{i=1}^{k-1} \frac{C_i}{min(p_i, D_i)} + \sum_{i=m+1}^{n} \frac{C_i}{min(p_i, D_i)} \leq 1. \tag{6.6}$$

Since the DVS scheduler can schedule any set of tasks schedulable by EDF, T_{new} can be admitted to the system without affecting feasibility as long as (6.6) is true for

all $k = 1, 2, ..., m + 1$. □

A naive implementation of the instantaneous density test algorithm would compute each of the sums in $O(n)$ steps for all $O(n)$ active jobs for an overall complexity of $O(n^2)$. However, if $O(n)$ storage space is available, the sums can be computed and accumulated in the array in $O(n)$ time.

The instantaneous utilization test provides a sufficient test for schedulability for any DVS algorithm that can schedule a set of jobs schedulable by EDF. Though the added utilization of the new task will necessarily increase energy consumption, the DVS algorithm can continue to create feasibly schedules as long as it recomputes how to stretch jobs after the new task is added.

Algorithms like CCEDF and and LAEDF require only that the utilization be updated to reflect the new task being added to the system. Feedback EDF requires more extensive changes because it relies on a precomputed table of available slack for future jobs. When a new task is added, the future slack table must be recomputed and slack reservation must be redistributed to existing jobs.

6.2.2 Feasible Deadline Computation

In some cases T_{new} cannot be scheduled feasibly as soon as it arrives, but it can be scheduled after enough of the work deferred by the DVS algorithm is completed. If the application can tolerate the first job completing later than specified by the task's relative deadline, we would like to compute the minimum deadline for which the first job and all subsequent jobs in T_{new} can be feasibly scheduled.

Algorithm 3, the Feasible Deadline Computation algorithm determines the minimum $D_{new,1}$ that will allow T_{new} to be admitted. The scheduler can produce a feasible schedule as long as the instantaneous utilization is less than or equal to one for all intervals, as discussed above. Thus, we need to find the last interval such that instantaneous utilization with the newly added task is no greater than one for all subsequent intervals. After this interval I_p the new task can be scheduled as a periodic task and no deadline will be missed.

$J_{new,1}$ can be treated as a sporadic job. Slack time in each interval prior to the $J_{new,1}$'s deadline is used for it's execution. The earliest interval I_{q+1} containing $d_{new,1}$ is thus decided such that $J_{new,1}$ can just finish its execution within intervals prior to I_q. The deadline of first job in the new task $d_{new,1}$ is set to the earlier of the end of I_p and I_q, whichever is later.

Algorithm 11 computes the deadline of $J_{new,1}$ by first sweeping forward through

intervals, starting with the first interval, to compute the sum of slack time available within each interval. The sweep stops in interval I_{fwd} when the total slack time is no less than the worst case execution time of $J_{new,1}$. It then sweeps backward from the interval after the last deadline of any active task, I_{m+1}, to the latest interval that has been visited by the forward sweep. The interval I_{bwd} chosen by the backward sweep is the first interval whose density exceeds one when added to the new task's density. The end of the later of I_{fwd} and I_{bwd} is chosen as $d_{new,1}$, so that T_{new} can be feasibly scheduled with the existing tasks.

Algorithm 11 Feasible deadline computation

Require: $p_{new}, C_{new}, D_{new}; C_{left,i}, \Delta_i \forall i : 1 \leq i \leq m+1; d_i, C_i, \forall i : 1 \leq i \leq n$
Ensure: $d_{new,1}$
  $d_0 = t$
  $slack \Leftarrow 0$
  $i \Leftarrow 1$
  while $i \leq m$ and $slack < C_{new,1}$ **do**
    $slack \Leftarrow slack + (1 - \Delta_i)(d_i - d_{i-1})$
    $i \Leftarrow i + 1$
  end while
  if $i = m + 1$ and $slack < C_{new,1}$ **then**
    $d_{forward} \Leftarrow \frac{C_{new,1} - slack}{1 - \Delta_{m+1}} + d_m$
  else
    $d_{forward} \Leftarrow d_{i-1}$
  end if
  $j \Leftarrow m + 1$
  while $j \geq i$ and $(\Delta_j + \Delta_{new}) \leq 1$ **do**
    $j \Leftarrow j - 1$
  end while
  $d_{backward} \Leftarrow d_j$
  $d_{new,1} \Leftarrow max(d_{forward}, d_{backward})$

The computation of deadline use the algorithm above requires $O(n)$ time. It is assumed the density for each interval is known. As we discussed in section 6.2.1, the computation of interval density can be finished in $O(n)$ time, if $O(n)$ storage space is available. Thus, the overall computation time for the algorithm is $O(n)$.

6.3 Summary

Ordinary admission tests are not sufficient when used with DVS because the scheduler may defer too much work to allow a new task to be scheduled. This work is not accounted for in the standard task admission tests. As with admission tests for

schedulers without DVS, our test has $O(n)$ time complexity, where n is the number of tasks in the system.

We have presented an online admission tests and deadline computation algorithm for adding periodic tasks to systems using real-time DVS scheduling. The first provides sufficient conditions for admission with any DVS algorithm capable of scheduling any task set that is schedulable by EDF, for example LAEDF [31]. The second algorithm computes a feasible deadline for the first job in the new task by which the new task can be admitted by the system.

The admission test determines whether the first instance of a job can run before a given deadline, and the subsequent jobs in the task will be schedulable with any DVS algorithm that can schedule any set of tasks schedulable by EDF. Simulations have been done to the admission and the feasible deadline computation. The detail of simulation results is given in Chapter 7.

Chapter 7: SIMULATION RESULTS

In previous chapters, we discussed distributed DVS-EDF scheduling algorithms, such as DSEDF, DCCEDF, ADCCEDF, DLAEDF and DFEDF, task assignment heuristics including Worst-Fit, Best-Fit, Communication-Aware Worst-Fit and MinΔP, and deadline assignment heuristics for distributed real-time system. In addition, a study of online admission tests for uniprocessor real-time system is discussed to deal with the dynamic task set having tasks arriving and leaving the system when system is running. These algorithms are evaluated using simulations. Simulation is a fast, flexible and generic way to evaluate and compare the performance of algorithms.

Before we describe the simulation results for each set of algorithms, the design of an Event-Driven Real-Time Simulator, EDRTSim, is given in the first section in this chapter. All the simulation results discussed later are based on the simulations done on this simulator. There are four groups of simulations done to evaluate the performance of algorithms, namely, task assignment simulation, deadline assignment simulation, distributed DVS-EDF simulation, and admission test for dynamic task set simulation. The detailed discussion of the simulation results is given in the sections following the description of simulator design.

7.1 Simulator Design

A survey has been done of the existing real-time simulators. The architecture-level simulators [10, 44, 7, 27, 6, 35] are assumed to be more accurate in algorithm performance estimation. Simulators, such as SimpleScalar [10], SimplePower [44] and Wattch [7], are only applicable to uniprocessor systems. Architecture-level multiprocessor system simulators, such as SimOS [35], Proteus [6] and LIMES [27], do not estimate system energy consumption. None of above simulators includes voltage scalable processor model. A high-level event-driven simulator is developed instead to simulate the algorithms for distributed real-time systems.

The event-driven simulator, EDRTSim, developed based on all the assumptions discussed in Chapter 3. This simulator takes the end-to-end task set as its input, assigns subtasks onto processors, schedules tasks according to specified real-time scheduling algorithm and produces the energy consumption of the processors and communications as the output. The block diagram of EDRTSim is shown in Figure 7.1.

Each of the processor module in the figure is composed of a local task set, a

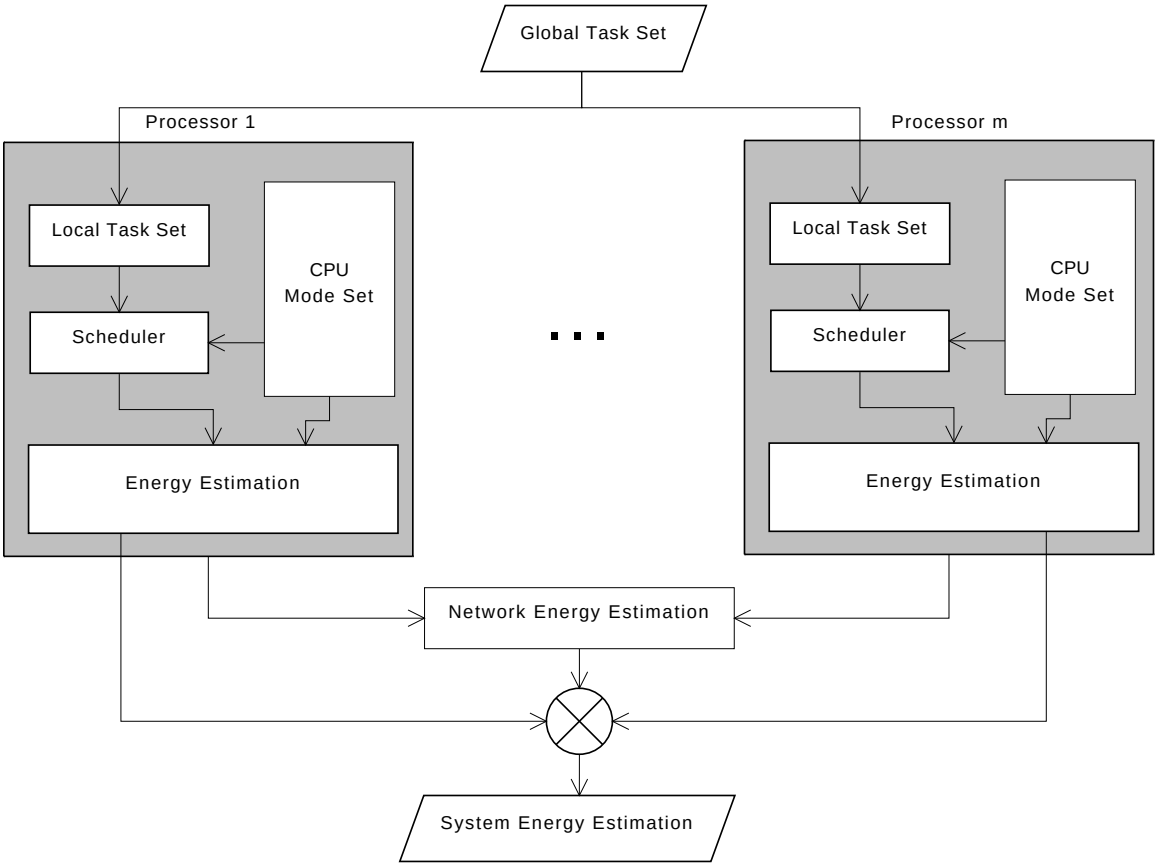


Figure 7.1: Block diagram for EDRTSim simulator

CPU mode set, a scheduler, and an energy estimator. The CPU mode set contains the information provided by a table, which is used by scheduler to make decisions on processor's speed when scheduling jobs in tasks. The energy estimator within the processor module estimate the energy consumed by job execution. The processor energy consumption is estimated using the sum of products of processor power consumption and job's execution time. Both of the power consumption and job's execution time are function of processor speed used to run the job.

$$E_{proc} = \sum_{\Delta t_i \in [0, t_{end}]} P(f_i) \times \Delta t_i(f_i) \tag{7.1}$$

Where t_{end} is the specified simulation end time. $P(f_i)$ is the power consumption when the CPU is operating at speed of f_i. The time interval used to execute a job at the speed of f_i is denoted as Δt_i.

The network energy estimator module computes the energy consumption based on the amount of Kbytes in message/data that have been transmitted through the network according to the assumptions we have made in Chapter 3.

$$E_{network} = \sum_{i} E_{Kbyte} \times nKbyte_i \tag{7.2}$$

Where E_{Kbyte} is the energy cost for transmitting unit Kbyte of message through the network. The variable $nKbyte_i$ is the number of Kbytes transmitted within a message or datum. The total number of messages or data transmitted through the network depends on how the subtasks are assigned among the processors. There is no energy cost on the network for the synchronization of two adjacent subtasks if and only if they are assigned to the same processor.

The detailed design of scheduler module and task set modules in EDRTSim is discussed in the following subsections.

7.1.1 Task Set And Task Class Hierarchy

There are two types of task set in EDRTSim, global task set and local task set. *Global task set* is composed of end-to-end tasks. These tasks provide information for task assignment and deadline assignment. Task assignment assigns subtasks from all tasks in the global task set onto processors. The set of subtasks assigned to a processor is the *local task set* for that processor. The task in the local task set behaves like a periodic task with a period, deadline and WCET. Local tasks on one processor may communicate with those on another processor based on the information provided by

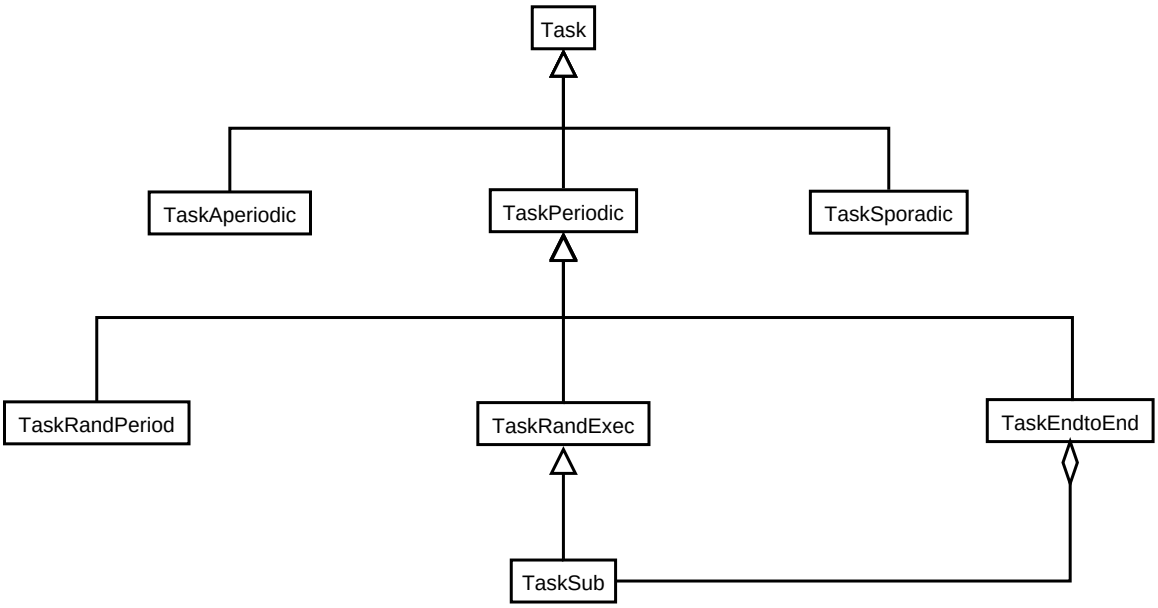


Figure 7.2: Real-Time task class hierarchy

the global task set.

As the input of the simulator, the global task set is generated randomly given a total system density. Each task set contains twenty end-to-end tasks taken from one of three categories with equal probability. Short tasks have a period of 1 ms to 10 ms, medium tasks a period of 10 ms to 100 ms, and long tasks have periods from 100 ms to 1000 ms as described by Pillai and Shin [31]. The number of subtasks in each of the end-to-end task varies from one to five. The worst-case execution time of each subtask is generated randomly based on its period and the required system density.

In order to handle different types of real-time tasks, EDRTSim creates a class for each type of real-time task using object-oriented programming language. The class hierarchy of the real-time task class is shown in Figure 7.2.

Class `Task` is an abstract class containing properties that all the real-time tasks share. `TaskPeriodic, TaskAperiodic` and `TaskSporadic` are three basic types of real-time task derived from `Task`. The dissertation focuses on the periodic tasks that have been discussed in Chapter 2. In practice, either the period or the execution time of a periodic task might vary within a range. `TaskRandPeriod` and `TaskRandExec` are two classes derived from `TaskPeriodic` to handle the above two types of real-time tasks. Real-time DVS algorithms take advantage of the periodic tasks with varying execution time by exploiting their dynamic slack time.

The end-to-end task has period and an end-to-end deadline. It can be considered

Table 7.1: Assumed processor operating points for PROC1

Voltage (V)	Relative Frequency	Power Consumption (Watt)
5	1.00	25.00
4	0.75	12.00
3	0.50	4.50
0	0.00	0.00

as a periodic task with a list of subtasks. Class `TaskEndtoend` is created as a child class from class `TaskPeriodic`. After a subtask from the end-to-end task is assigned with a deadline and allocated to a processor, it can be treated as a periodic task with varying execution time. Class `TaskSub` contains all the properties inherited from class `TaskRandExec` and information needed for data and message transmission between processors. Class `TaskSub` is included in class `TaskEndtoend` because that the end-to-end task is composed of a chain of subtasks.

7.1.2 Processor and Network Model

The processor in the simulator EDRTSim is modeled in a look up table containing the processor's operating voltages, frequencies and power consumptions with a specified voltage. The scheduler is capable of selecting the appropriate processor operation voltage and frequency that satisfies the timeliness of the real-time schedule. The energy of task execution is computed by the simulator according to the power consumption of the processor in its look up table given the operating frequency using Equation 7.1.

There are two different processor models used in the simulations to reveal the effect of different processors on the performance of specified algorithms. The first processor model (PROC1) assumes that each processor has three relative operating frequencies [31]. The corresponding voltage and power consumption are given in Table 7.1, based on Equation 2.1 and Equation 2.2.

Another processor model is based on measurements of an IBM PowerPC 405LP processor [30]. The measurements evaluate the power consumption of an SOC design using PowerPC 405LP, a dynamic voltage scalable embedded processor. Table 7.2 gives the operating points of PowerPC 405LP processor.

EDRTSim takes account the energy consumed by communication between dependent subtasks using Equation 7.2. The unit communication energy cost, E_{Kbyte}, is the amount of energy consumed to transmit every 1 Kbyte of data. The size of messages transmitted between any two dependent tasks are uniformly distributed between 5 KB

Table 7.2: Operating points for PowerPC 405LP

Voltage (V)	Frequency (MHz)	Power Consumption (mW)
0.18	100	27.68
0.75	180	112.55
1.05	266	232.47
1.42	333	313.65
1.90	398	500.00

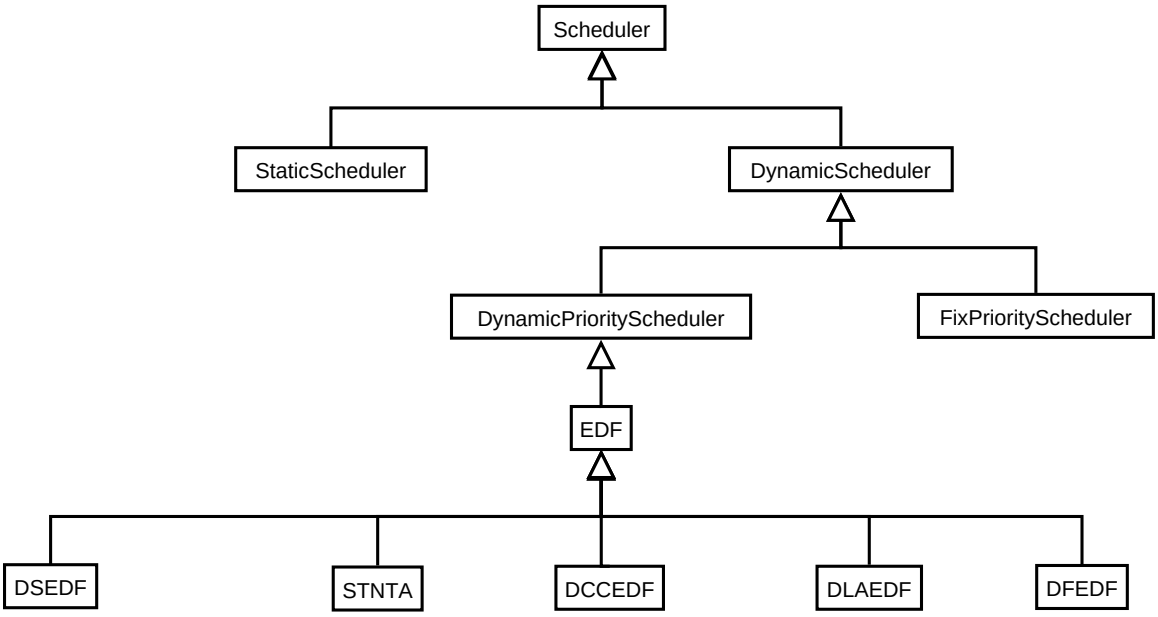


Figure 7.3: Real-Time scheduler class hierarchy

to 20 KB.

7.1.3 Event-Driven Scheduler Design

The scheduler in EDRTSim is capable of implementing various scheduling algorithms. Figure 7.3 shows the class hierarchy for the scheduler classes that are used by the simulator. The class `Scheduler` is an abstract class. Although this class cannot be instantiated to schedule any task, it provides a foundation for all the other concrete scheduler classes. The hierarchy of scheduler is built according to the classification of real-time schedulers discussed in Chapter 2. We focus on the design of EDF based schedulers. The DVS-EDF scheduler classes, such as `DSEDF`, `STNTA`, `DCCEDF`, `DLAEDF` and `DFEDF`, are classes created for each of DVS-EDF scheduling algorithms discussed in Chapter 4.

The design of real-time schedulers in our distributed system is based on six events

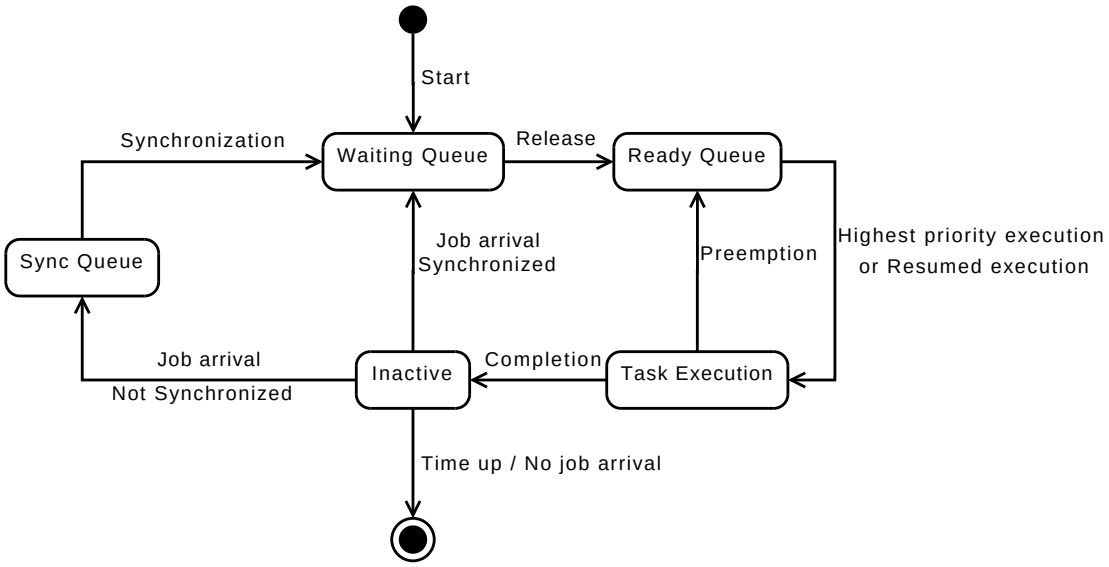


Figure 7.4: Finite-state diagram for a job in a RT task

that happens to a job running on the processor. The six events are job arrival, processor synchronization, job release, scheduled execution, job completion and job preemption. Each of the events results in a job's state transition. Jobs of each task running on the processor assume five states, waiting for release, ready to run, execution, inactive and waiting for synchronization. The state transition of a task is shown in Figure 7.4.

To implement the state transitions of a task, two priority queues are set up for the scheduler simulator, waiting queue and ready queue. Waiting queue contains all the tasks waiting for release ordered according to their earlist release time. While the ready queue lines up all the jobs released from the waiting queue in an order such that the job with the earlier absolute deadline positioned closer to the head of the ready queue.

Besides the priority queues, there is a linked list, synchronization queue, used to keep the information about jobs that are waiting for synchronization. As we have discussed in Chapter 3, the release guard synchronization protocol is implemented in EDRTSim. To maintain the dependency between subtasks within an end-to-end task when the adjacent subtasks are assigned to two different processors. The successor subtask has to wait for the message and/or data transmitted from its predecessor on the other processor before it can execute. A job is added to the end of the synchronization queue when the job in that task is arrived before the its synchronization happens. Tasks are moved from the list to the waiting queue when their synchronization message is received.

Initially, all the jobs from each subtask on the processor are added to either waiting queue or synchronization queue based on whether the subtask is the first in its parent end-to-end task. The first job from the first subtask is put directly in the waiting queue since there is no synchronization needed. The other jobs are linked to the list waiting for synchronization. When the release time of the task in the waiting queue is reached, the scheduler release the task from waiting queue to the ready queue. The task in the head of ready queue is scheduled to run for a period of time with a specified processor speed.

A decision has to be made by the scheduler on how fast the job in the head of ready queue can run. The decision is done by DVS-EDF scheduling algorithm instantiated on the processor. The EDF scheduling without DVS schedules all the jobs running at processor's full speed. How the decision is made for the DVS-EDF scheduling algorithms listed in the Figure 7.3 has been discussed in Chapter 4.

The length of time for job execution depends on the time upon job's execution completed and the nearest release time of jobs in the waiting queue, whichever comes sooner. The job's execution is simulated without actually running the job. EDRTSim "executes" the job by advancing the processor time according to the scheduled execution time and the processor's speed. If the job cannot be completed before another job is released from the waiting queue, a priority comparison has to made between these two jobs. The job with the earliest absolute deadline has highest priority to run. If the new released job wins the competition, a preemption happens. The running job is stopped and put back to the ready queue. Preemption and resumption can happen multiple times until a job is finally completed.

Upon completion of the job in a subtask, a synchronization message is sent to the successor of the subtask. The completed task becomes inactive until the next job in the task is released. The new job is added to waiting queue if its predecessor completes, or to synchronization list if no synchronization message for this subtask is received. The simulator stops when a predetermined end time is reached.

7.1.4 Summary

The event-driven real-time simulator EDRTSim is a high-level simulator that can achieve the simulation requirements of DVS-EDF scheduling algorithms discussed in this dissertation. EDRTSim can simulate partitioned distributed real-time systems with one or more processors. It supports different processors with information of their working frequencies and power consumptions corresponding to each frequency.

This event-driven simulator is fast than the architecture-level ones because it need not simulate execution at instruction level. Statistics or trace results of tasks can be used to generate task execution times used in EDRTSim. The accuracy of energy consumption can be calibrated by modify the processor model, task model or network model.

EDRTSim is extendable for more complicated real-time system simulation. The object-oriented design makes it easy to add additional modules such as memory system, I/O system and hard disks to the simulator. The idea of processor can be extended to any CMOS devices. That is EDRTSim can be modified for heterogeneous real-time system simulation. This simulator can support more scheduling algorithms simply by creating and adding a new scheduling class to the scheduler class hierarchy.

The drawback of this event-driven simulator is its low estimation accuracy compared with cycle-accurate architecture-level simulators. The accuracy of the energy estimation of EDRTSim, however, is accurate enough to evaluate the performance of all the DVS-EDF scheduling algorithms discussed in this dissertation.

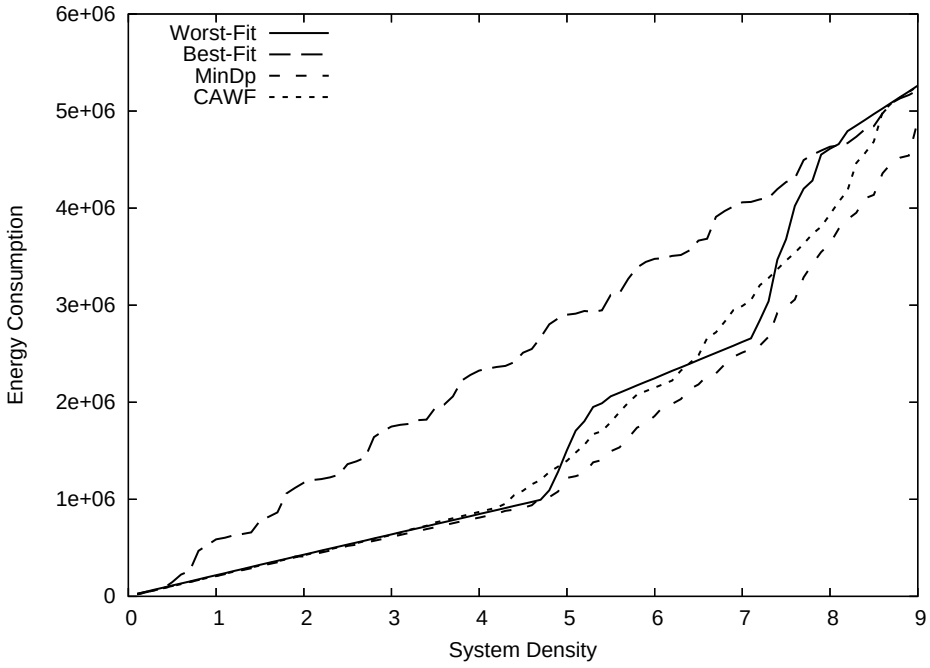
7.2 Task Assignment Simulation

Simulations have been done to compare the power-conservation performance of the Best-Fit, Worst-Fit, Communication-Aware Worst-Fit, and MinΔP task assignment heuristics. In the simulation, the processor model PROC1 is used. The actual execution time of each job in the simulation is a randomly generated value between zero and its task's worst case execution time. Actual execution times have a modified Gaussian distribution with a mean of one half of the task's WCET. Values greater than WCET are clipped to the WCET and values less than 1 % of WCET are limited to 1 % of WCET.

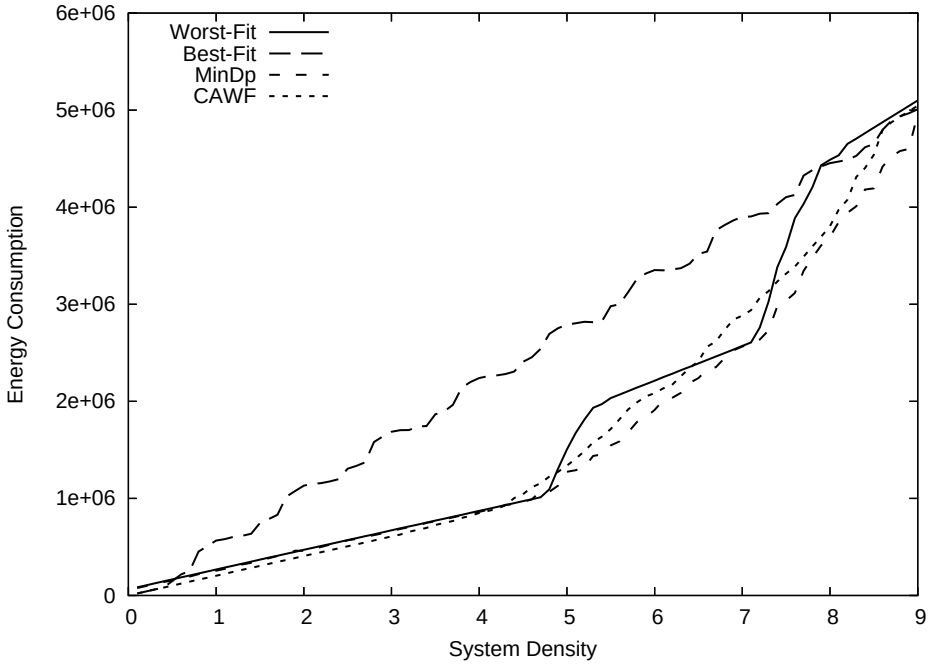
7.2.1 Task Assignment Comparison

The first set of simulations compares the energy-conserving performance and feasibility performance of task assignment algorithms discussed in Section 5.

Figure 7.5 shows the comparisons between Worst-Fit, Best-Fit, CAWF, and MinΔP for a system with 10 processors scheduled with DSEDF. The X-axis indicates the system's total density, while the Y-axis measures the total energy consumption. The four curves in each graph depict the energy consumption for tasks assigned with each task assignment algorithm. The comparison in Figure 7.5(a) is with a unit communication energy, E_{Kbyte}, of 0.01 mJ/B, and Figure 7.5(b) has E_{Kbyte} equal to 0.10 mJ/B.

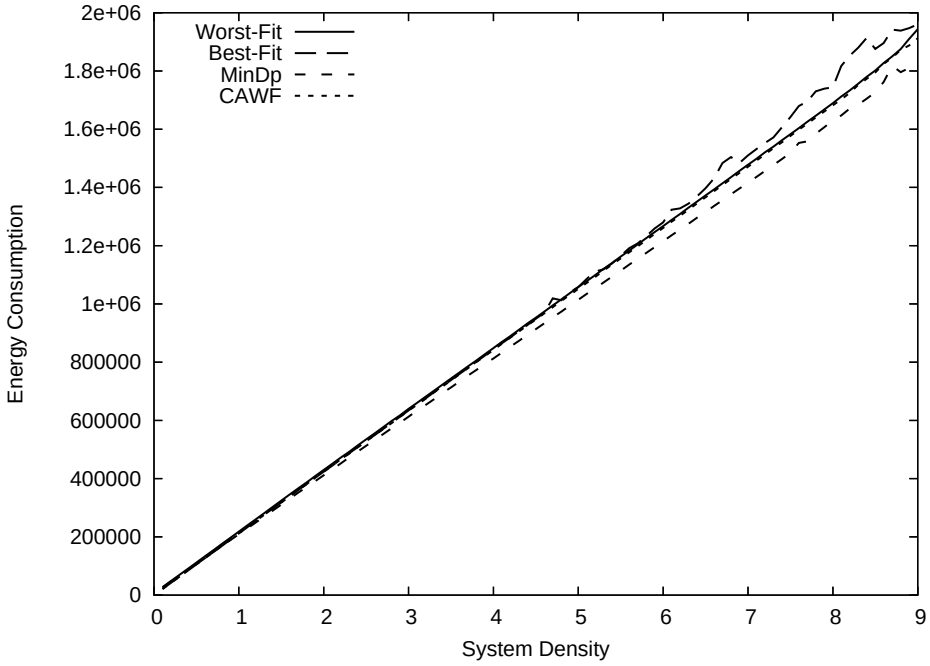


(a) $E_{Kbyte} = 0.01\,\mathrm{mJ/B}$

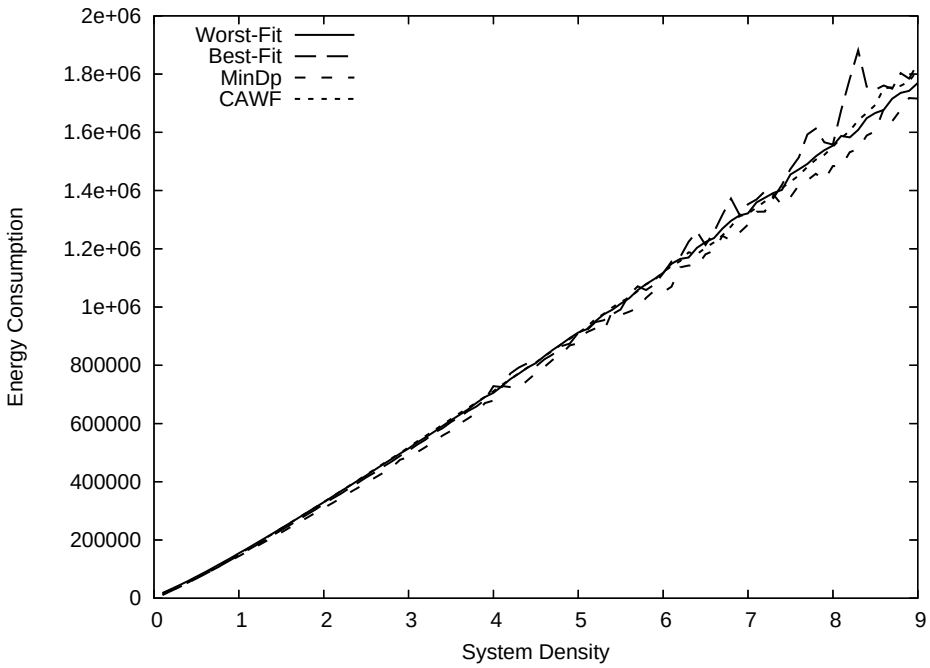


(b) $E_{Kbyte} = 0.10\,\mathrm{mJ/B}$

Figure 7.5: Task assignment with DSEDF



(a) DLAEDF Scheduling



(b) DFEDF Scheduling

Figure 7.6: Task assignment performance comparison ($E_{Kbyte} = 0.01\,\mathrm{mJ/B}$)

For DSEDF the Best-Fit assignment results in the highest energy consumption. By assigning as many as tasks as possible to the most heavily loaded processor, Best-Fit requires higher operating frequency in order to maintain schedulability on processors with assigned tasks. Though some processors may be idle, static power is low enough that running more processors at a lower speed is more efficient than fewer processors at higher speeds. Systems with higher static power may benefit more from Best-Fit depending on the relative weights static and dynamic power.

A jump in the energy consumption with the Worst-Fit happens when the evenly distributed workload on each of the processors requires a higher CPU operating speed. This jump occurs first near a system density of 5 and again near 7.5, corresponding to the relative frequencies of 0.50 and 0.75 in Table 7.1.

The Communication-Aware Worst-Fit algorithm often has better energy-conserving performance than the Worst-Fit. The tendency to keep related subtasks together forces some processors to switch to a higher frequency sooner than with worst fit, for example for densities from about 4.5 to 5 and 6.3 to 7.2 in Figure 7.5(a). For other system densities, however, processor loads remained well balanced, and communication energy was reduced because subtasks of the same end-to-end task tend are assigned to the same processor when possible. In these regions, Communication-Aware Worst-Fit used less energy than Worst-Fit. The difference between Worst-Fit and Communication-Aware Worst-Fit is even more evident in Figure 7.5(b). The higher communication cost ratio makes the communication cost savings more evident.

The MinΔP algorithm has the best energy-conserving performance of any of the algorithms shown in Figure 7.5. MinΔP saves more energy than both the Worst-Fit and the Communication-Aware Worst-Fit because MinΔP makes task assignments based on how both processor and communication power consumption will change when a new task is assigned. At worst, MinΔP is just below the Worst-Fit energy consumption. After total system utilization reaches approximately 5, MinΔP makes more power-efficient task assignments than Worst-Fit, keeping tasks together when it will save on communication cost or prevent a processor from increasing its speed.

DLAEDF and DFEDF are able to produce much better energy savings than DSEDF because they use static slack and dynamic slack to slow down the task execution. Figure 7.6 shows how they perform with each of the four task assignment algorithms. The difference in total energy consumption between the four task assignment algorithms is very small with MinΔP having a small edge over the other three algorithms.

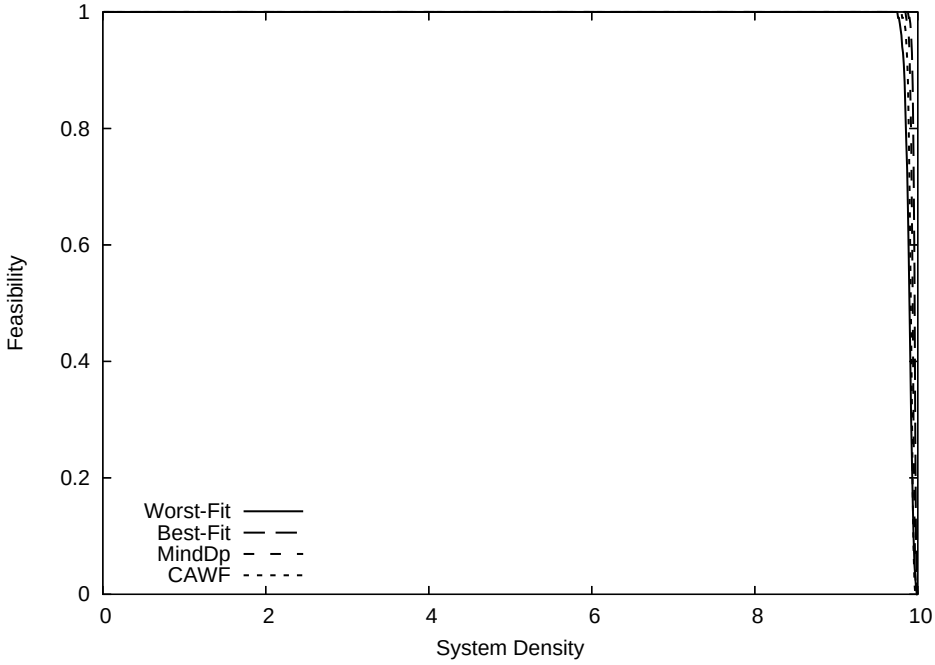
The small difference of energy consumption between task assignment algorithms

is due to the amount of dynamic slack available. Dynamic slack exists in the system when tasks have an actual execution time shorter than their worst case execution times. For the Gaussian distributed task actual execution time, the actual processor load is one half of the worst case processor load. Moreover, a subtask with siblings in an end-to-end tasks always has a deadline shorter than its period to allow all its sibling tasks to complete before the end-to-end deadline. As a result, processor density can be two to five times higher than the processor utilization. The versions of DLAEDF and DFEDF modified to work with deadlines shorter than their periods [41] are able to use the slack between a job's deadline and next release time to decrease processor frequency.

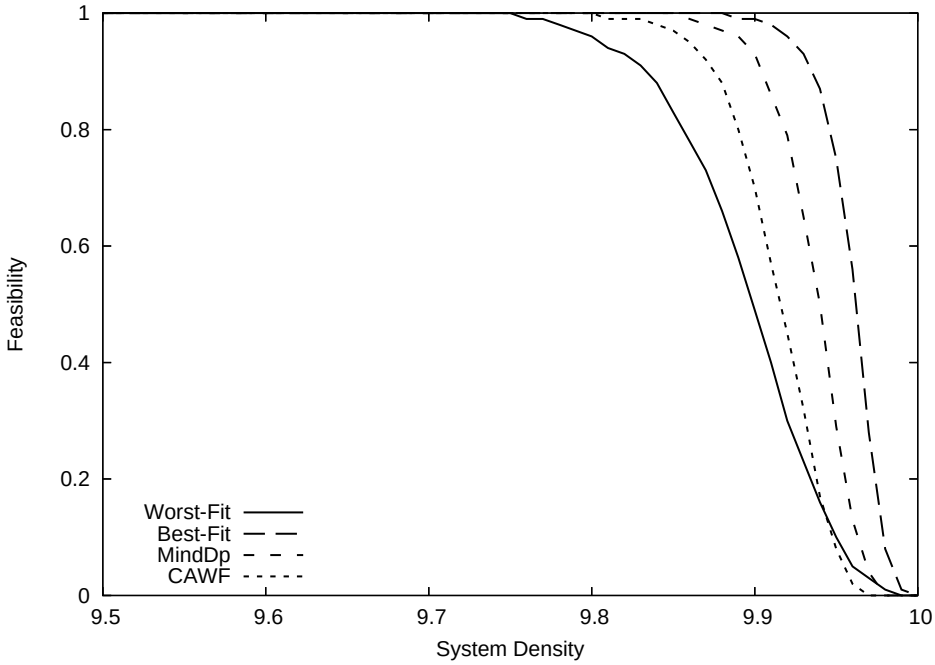
The task assignment heuristic affects the feasibility of distributed system as well. A real-time task set is said to be feasibly assigned to the distributed system if the tasks on each of the processor are schedulable using a given real-time scheduler. The feasibility performance of four task assignment heuristics is shown in Figure 7.7. The number in the X-axis stands for the system's density. Figure 7.7(a) shows the feasibility performance of each task assignment with the system's density varying from 0.1 to 10. The difference of the feasibility performance is better shown in Figure 7.7(b) with system's density varying from 9.50 to 10.00. The Y-axis indicates the feasibility performance of each task assignment heuristic. Feasibility performance is expressed as a percentage of task sets that can be feasibly scheduled by EDF when using a certain task assignment algorithm. EDRTSim simulator assigns 1000 real-time task sets randomly generated with different random seeds using each of the four task assignment heuristics, generating the percentage of feasibly scheduled task set. There are 50 tasks in each task set.

Best-Fit has the best feasibility performance shown in Figure 7.7. This heuristic assigns 100 % of the task sets successfully even when the system's density is as high as 9.88. Worst-Fit starts to fail when the density is 9.75, but CAWF has better performance than Worst-Fit when the system density is lower than 9.95. MinΔP performances better than CAWF, which results in 100 % feasibly task assignment with a system density up to 9.85.

MinΔP has better energy-conserving and feasibility performance than Worst-Fit or CAWF according to the simulation results presented above. Although the feasibility performance of Best-Fit is better than MinΔP, when scheduled using DSEDF, MinΔP saves more energy than Best-Fit. For aggressive DVS-EDF scheduling, such as DLAEDF and DFEDF, the difference in energy-conserving performance of MinΔP and Best-Fit is small. With a higher feasibility performance, Best-Fit would be a bet-



(a) Full scaled



(b) Enlarged area

Figure 7.7: System feasibility with task assignment heuristics

ter task assignment for system scheduled with DLAEDF or DFEDF.

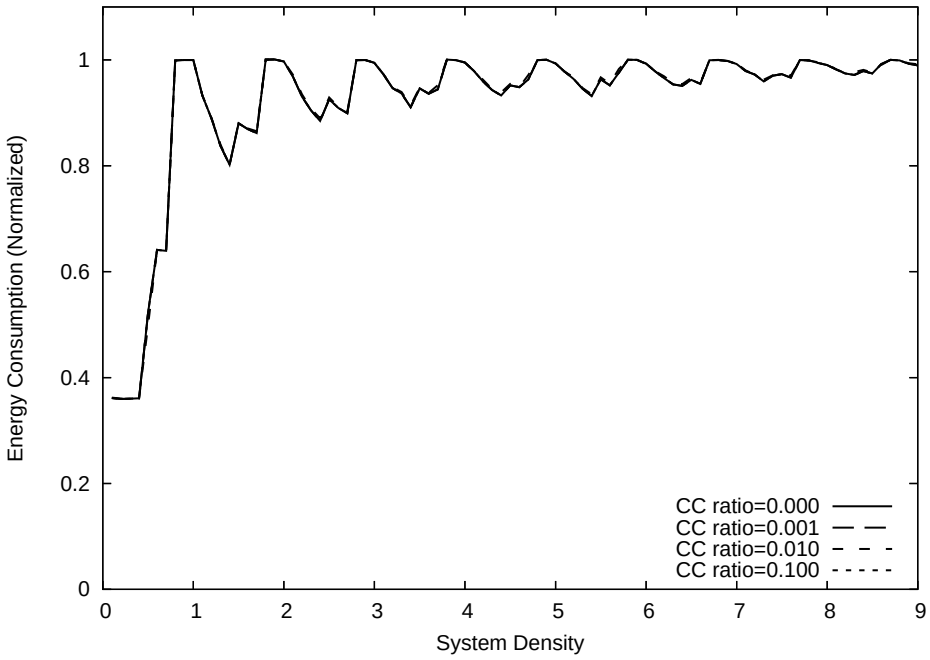
7.2.2 Effect of Communication Cost on Task Assignment

As seen in Figure 7.5, communication cost does have some affect on how much energy each task assignment algorithm requires. Figure 7.9 shows how communication cost affects each of the task assignment algorithms when DSEDF scheduling is used. For each of task assignment algorithm, four communication cost ratios are used in the system, 0.000 mJ/B, 0.001 mJ/B, 0.010 mJ/B, and 0.100 mJ/B. The X-axis is the system's density. And the Y-axis is the energy consumption with each communication cost normalized to that with EDF scheduling with the same task assignment and communication cost ratio.

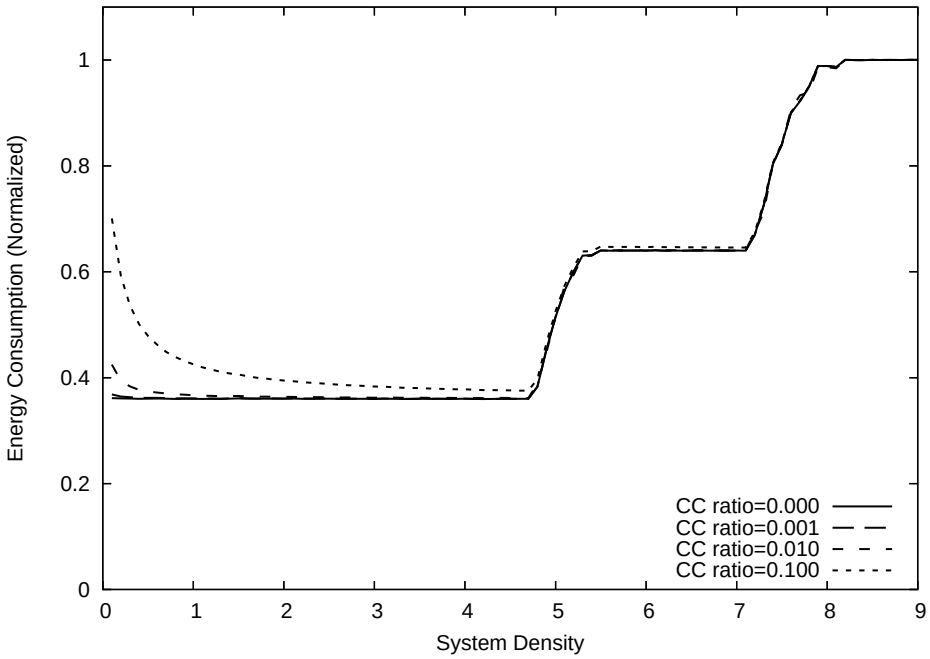
For the Best-Fit task assignment shown in Figure 7.8(a), the communication cost does not make any difference to its performance. Best-Fit squeezes as many tasks as possible, either dependent or independent, onto the same processor. Subtasks of the same end-to-end task are scheduled in sequence causing them to be likely to be placed on the same processor, which reduces the communications between tasks. When the system's density is very low (under 0.5 in Figure 7.8(a)), DSEDF with the Best-Fit is capable of saving as much as 63 % of EDF's energy consumption. As the system's density increases, the performance of DSEDF with the Best-Fit degrades rapidly compared with EDF because each processor with assigned tasks must run near full speed.

The performance of the Worst-Fit, shown in Figure 7.8(b), is affected by the communication among tasks. The greater the communication cost, the worse the performance when system's density is below five. Based on the end-to-end model, the density of dependent subtasks in an end-to-end task is equal when using proportional deadline assignment. In order to distribute the workload evenly among processors, the Worst-Fit is very likely to assign subtasks in the same end-to-end task on different processors, maximizing communication cost. The communication cost is significant comparing to the computation power consumption when the system's density is low, but becomes less significant as density increases.

Communication-Aware Worst-Fit greatly reduces the communication cost of worst fit by trying to keep subtasks in the same end-to-end task on the same processor while evenly distributing the workload. The changes in the system's communication cost do not make a visible difference Communication-Aware Worst-Fit's performance, shown in Figure 7.9(a).

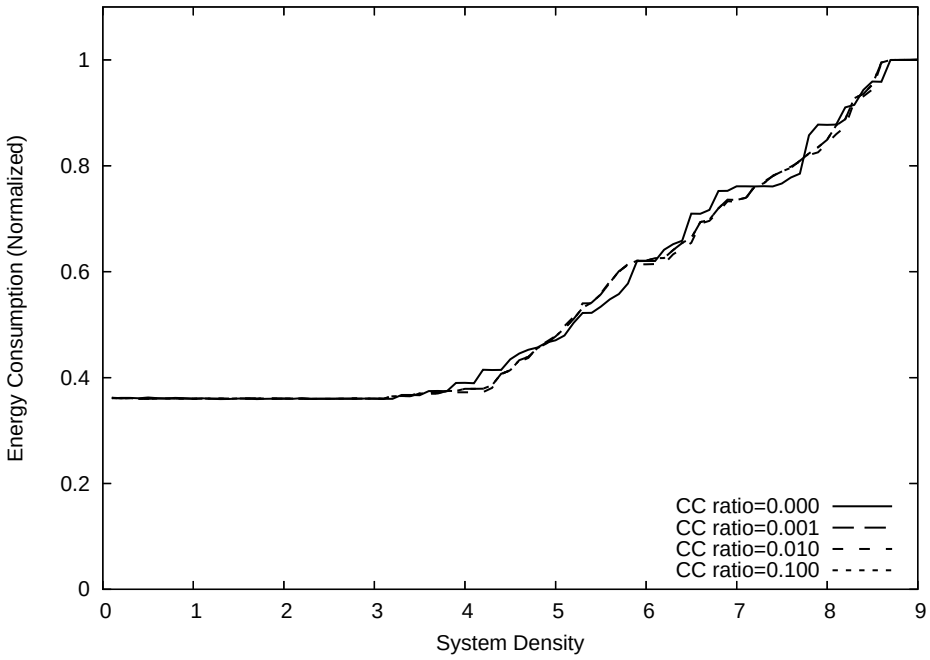


(a) Best-Fit

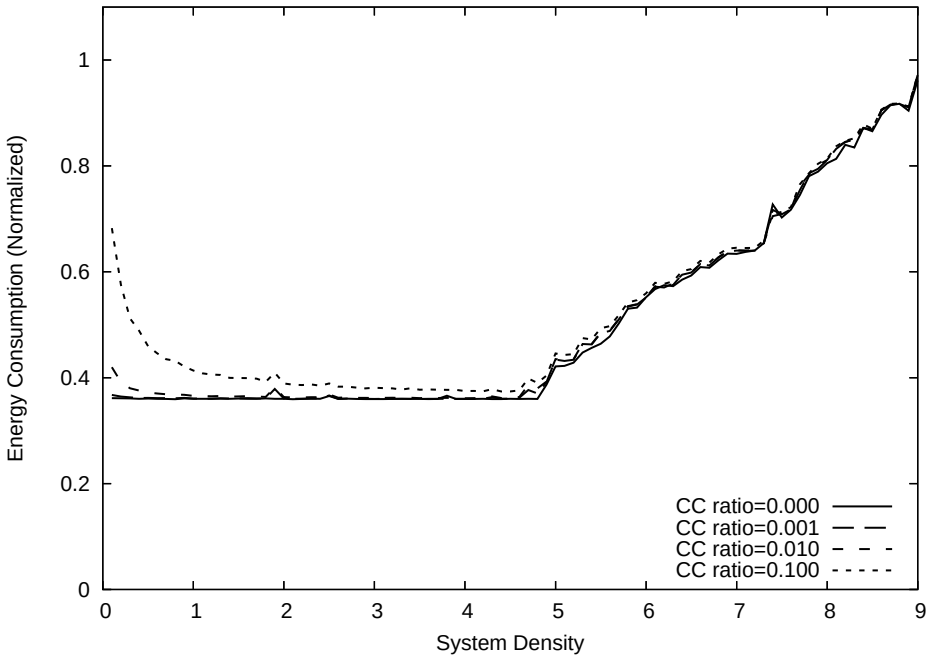


(b) Worst-Fit

Figure 7.8: Worst-Fit and Best-Fit with DSEDF



(a) Communication-Aware Worst-Fit



(b) MinΔP

Figure 7.9: CAWF and MinΔP with DSEDF

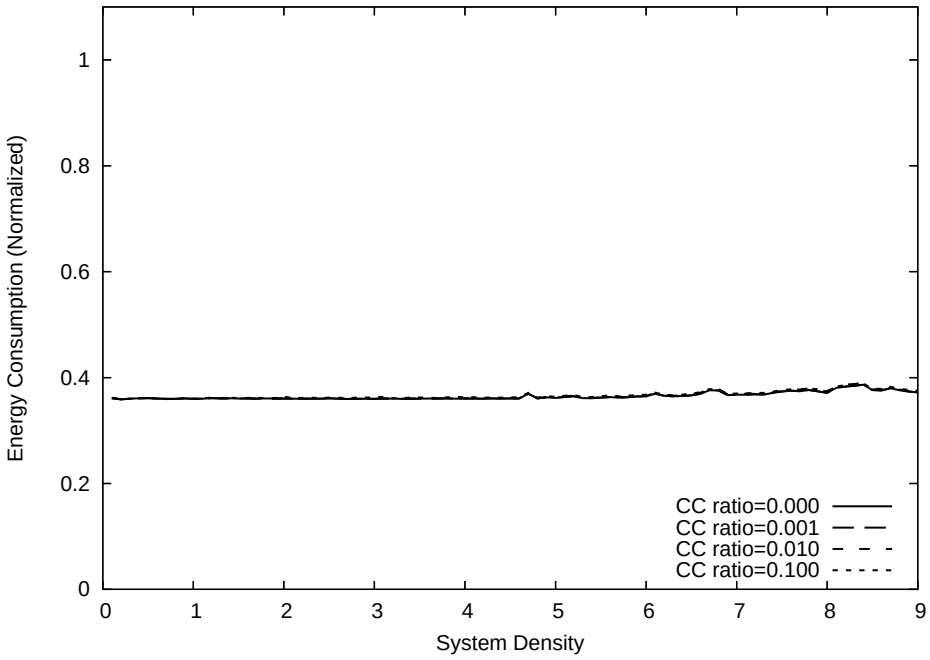
Figure 7.9(b) shows that MinΔP has a noticeable communication cost for system density below 5, very much like Worst-Fit. This communication cost is surprising since MinΔP directly accounts for communication energy. It is possible that communication cost is underestimated in relation to computation cost when system density is low. When system density is above four, tasks assigned by MinΔP clearly consume less energy than those assigned by other algorithms.

Similar results for the effect of communication cost effect on task assignment heuristics for DLAEDF and DFEDF are shown in Figures 7.10 through 7.13. Best-Fit with DLAEDF and DFEDF is not affected by the communication cost because this heuristic assigns as many as subtasks as possible onto one processor. The effect of communication cost is reduced to minimum and the communication energy cost is negligible in comparison with execution energy cost. The CAWF is not affected by system communication cost much as well. Instead of assigning all the task onto one processor as Best-Fit, CAWF trying to assign only the subtasks within the same end-to-end task onto the same processor while keeping the balance of workload among processors. Worst-Fit and MinΔP with DLAEDF and DFEDF is affected by the system communication cost.

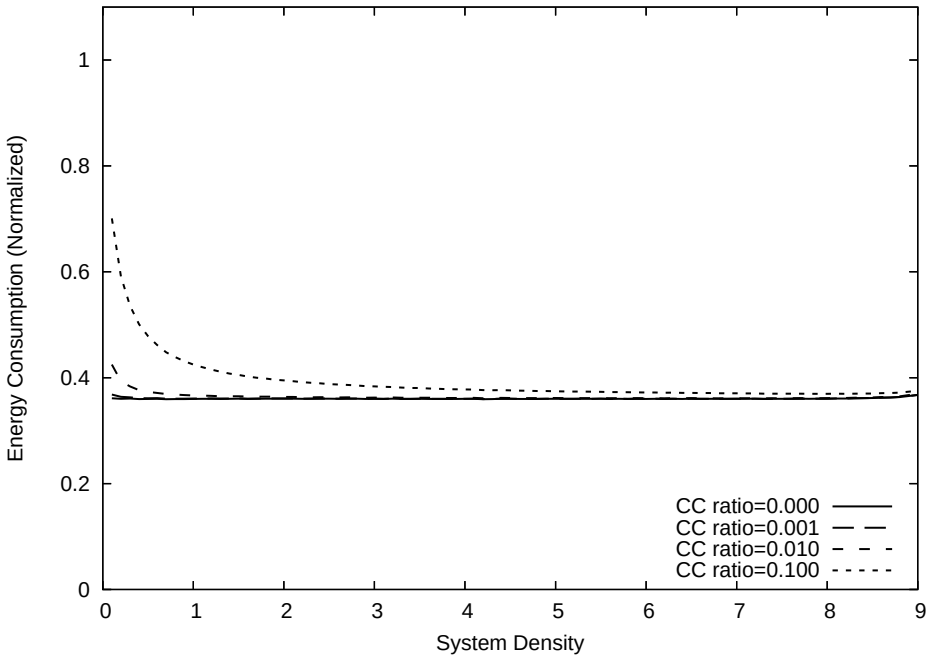
7.3 Deadline Assignment Simulation

Another set of simulations are done to compare the effect of each deadline assignment heuristic on the DVS-EDF energy saving performance. Ten homogeneous processors using the PowerPC processor model is used in the simulation. Each of the subtasks is assigned to the processor using MinΔP task assignment. The unit communication energy cost, E_{Kbyte} is set to 0.01 mJ/B. The task's actual execution time is generated randomly according to modified Gaussian distribution with the standard deviation of 1 ms. Random values below 1 % of WCET are limited to 1 % of WCET, and random values greater than WCET are limited to the WCET. Mean of each subtask's AET is randomly generated between the task's minimum execution time and its WCET. As shown in Section 5.2, when the ratios of AET mean of subtasks within an end-to-end task are same, NPD and ANPD result in the same deadlines for each of the subtasks. The random mean of AET is used to show the performance difference of NPD and ANPD.

The deadline assignment simulation results are given in Figure 7.14 through 7.16. Each set of figures contains a two graphs showing absolute energy consumption and normalized energy consumption with three deadline assignments, PD, NPD, and

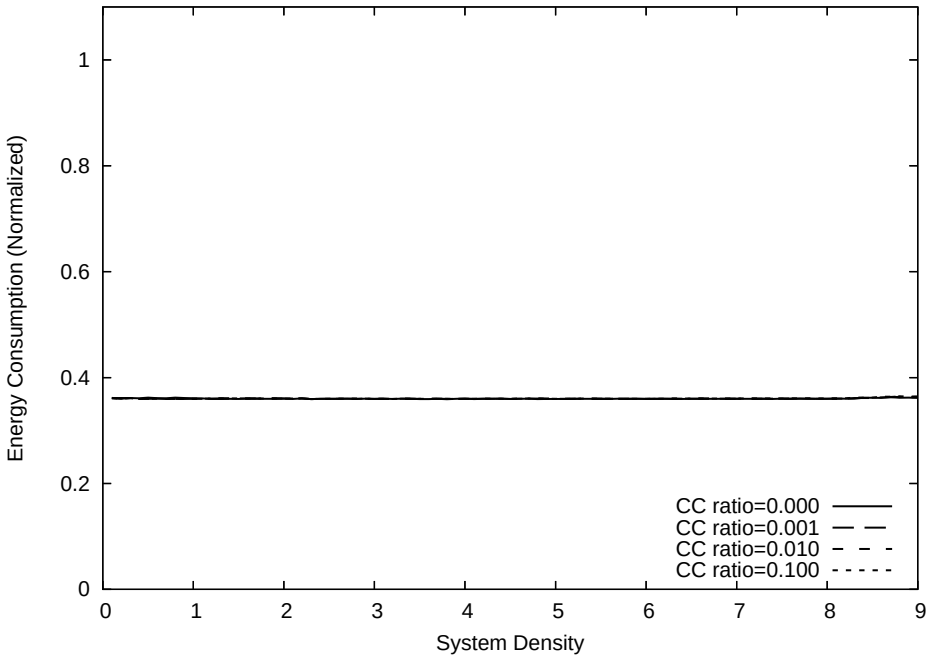


(a) Best-Fit

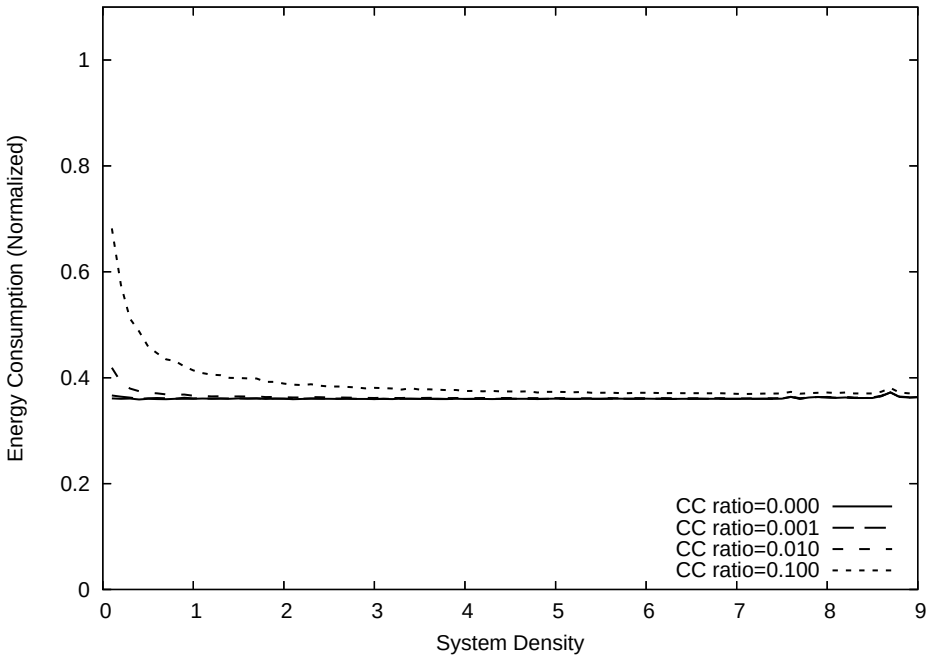


(b) Worst-Fit

Figure 7.10: Worst-Fit and Best-Fit with DLAEDF

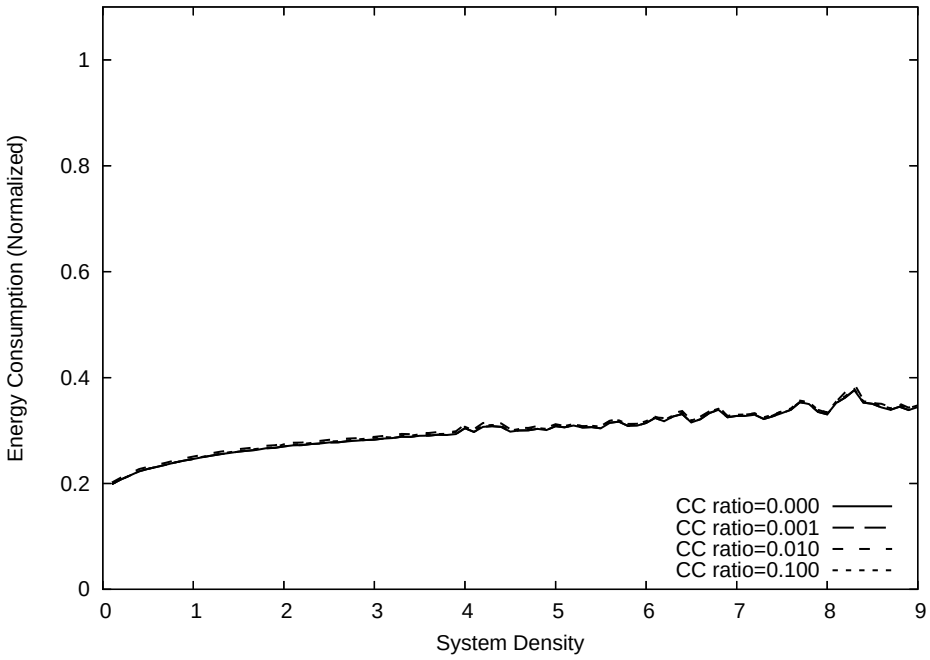


(a) Comm-Aware Worst-Fit

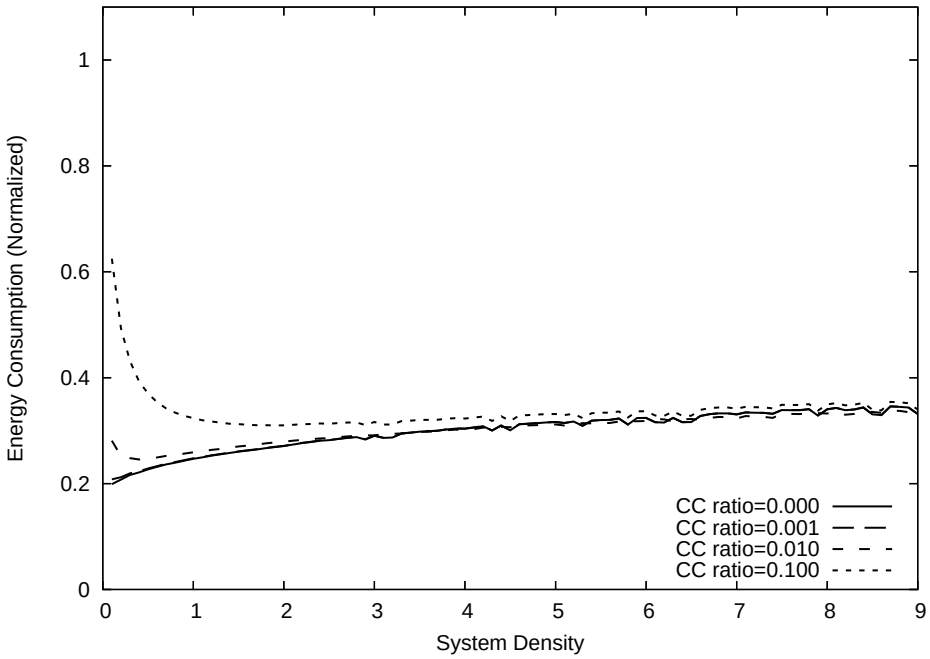


(b) MinDP

Figure 7.11: CAWF and MinΔP with DLAEDF

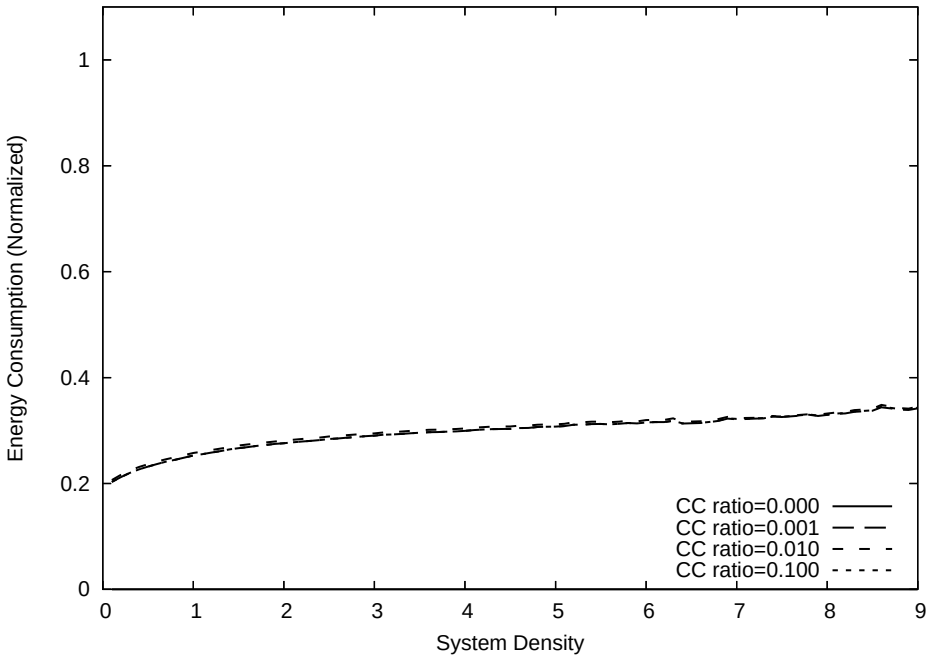


(a) Best-Fit

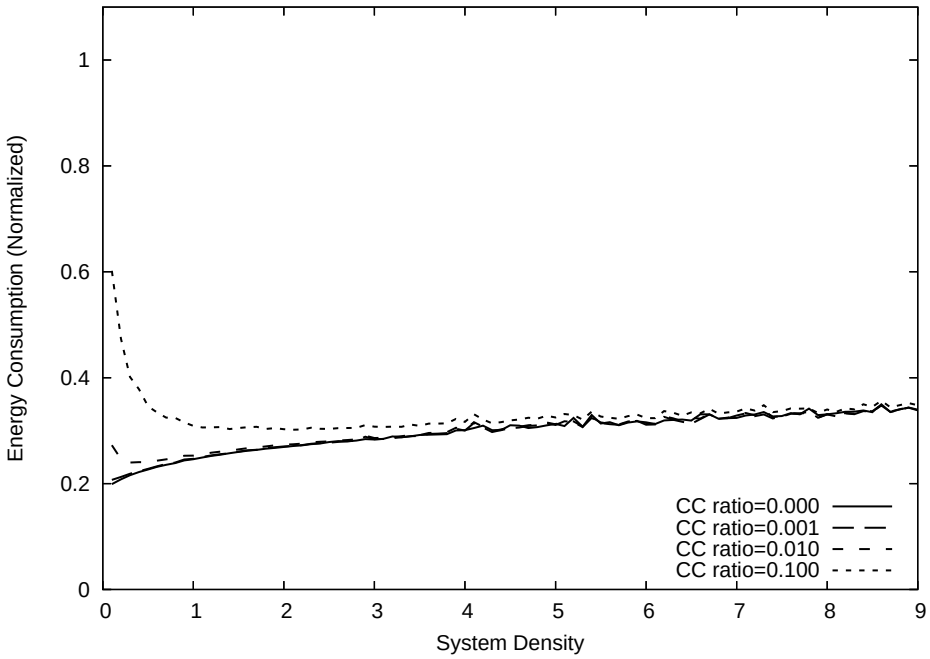


(b) Worst-Fit

Figure 7.12: Worst-Fit and Best-Fit with DFEDF

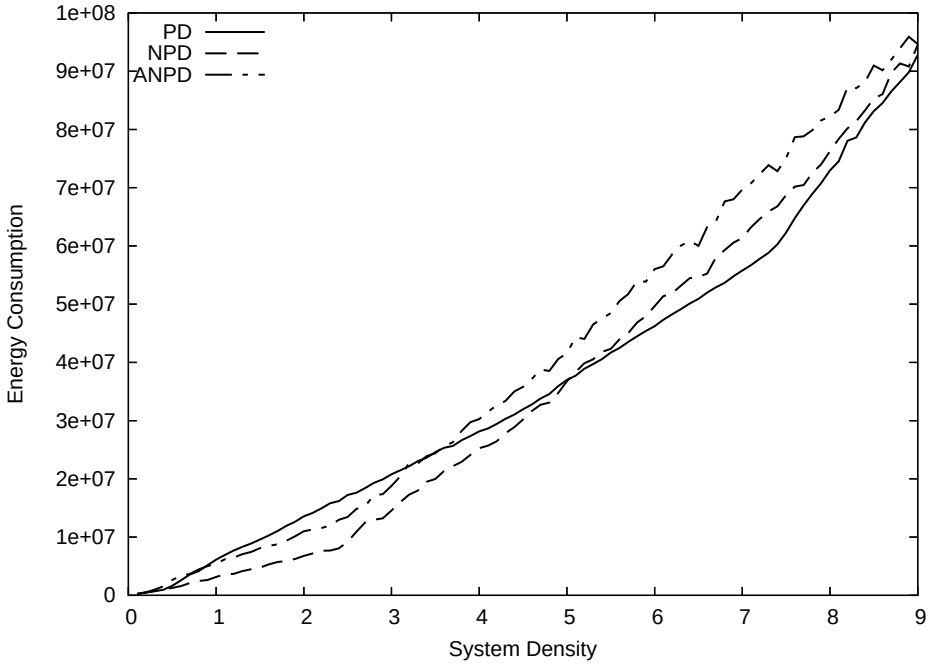


(a) Comm-Aware Worst-Fit

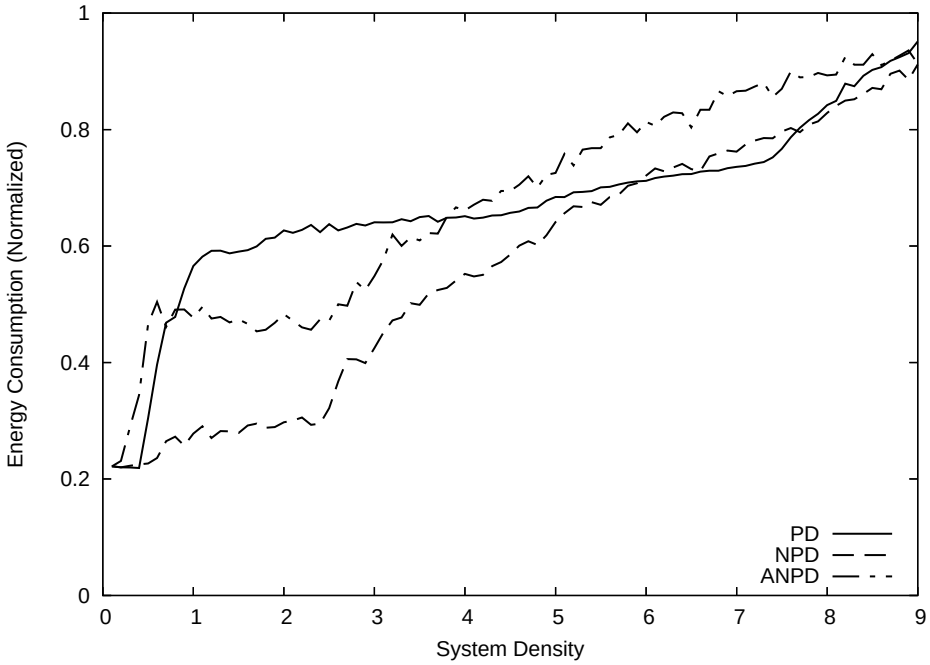


(b) MinDP

Figure 7.13: CAWF and MinΔP with DFEDF

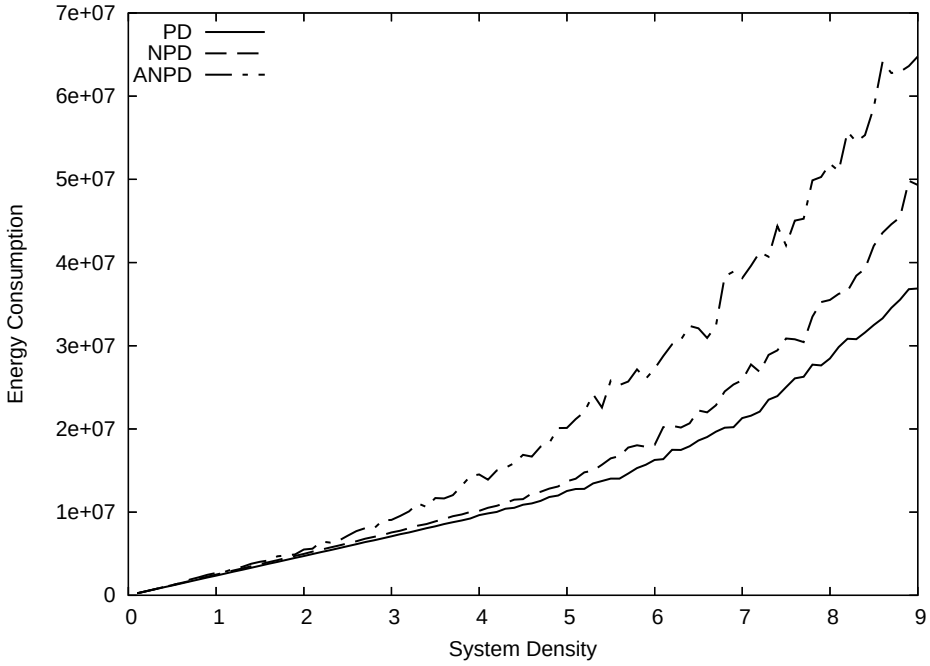


(a) Absolute energy consumption

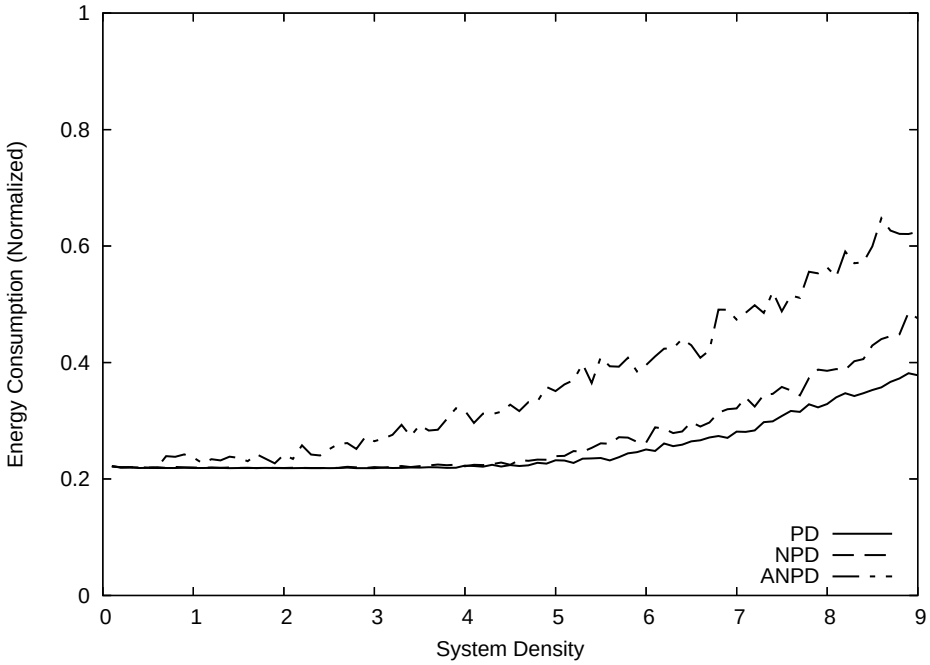


(b) Normalized energy consumption

Figure 7.14: Deadline assignment comparison with DSEDF

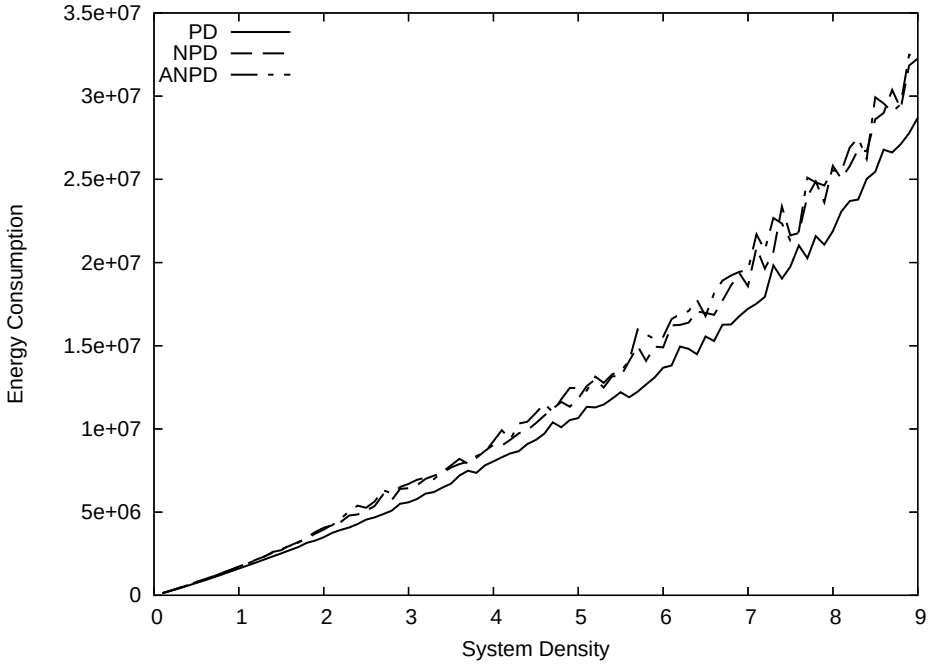


(a) Absolute energy consumption

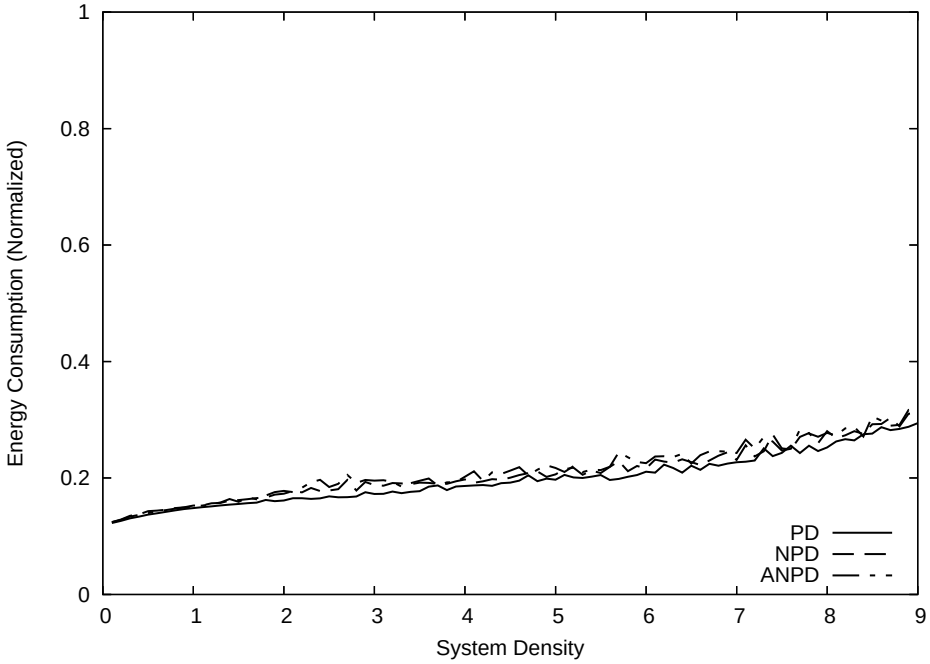


(b) Normalized energy consumption

Figure 7.15: Deadline assignment comparison with DLAEDF



(a) Absolute energy consumption



(b) Normalized energy consumption

Figure 7.16: Deadline assignment comparison with DFEDF

ANPD. The normalized energy consumption is the energy consumed by DVS-EDF scheduled system normalized to that consumed by EDF scheduled system. The X-axis in each of the figures is the total system density of the distributed system varying from 0.5 to 9.0. Figure 7.14, 7.15, and 7.16 contains the simulation results using DSEDF, DLAEDF, and DFEDF scheduling algorithm, respectively.

We observed missed local deadlines when using NPD and ANPD in the simulation. This happens because NPD and ANPD balance the workload among processors by assigning shorter local deadlines to subtasks on the lightly loaded processors and longer deadlines to those on the heavily loaded ones. It is possible for some subtasks on a lightly loaded processor be assigned with a local deadline very close to or even shorter than its WCET. The local deadline shorter than the subtask's WCET may cause deadline misses when scheduling with EDF-based scheduling algorithms.

DSEDF scales the processor's speed based on the density of that processor. The balanced workload is favored by DSEDF. When the system's density is low, the possibility of appearance of very short local deadline is low. Thus the performance of NPD and ANPD is better than PD. When the amount of very short local deadlines increases with the increase of the system's density, the performance of NPD and ANPD degraded by the increasing number of very short local deadlines. The very short local deadlines cause the processor's density very close to or even greater than 1. Full speed has to applied to such processors when scheduled by DSEDF. In the Figure 7.14(b), when the system's density is lower than 4.0, ANPD has saves a larger percentage of energy than PD. NPD performs better than PD when the system's density is lower than 6.

The PD performance best among three deadline assignment heuristics with DLAEDF and DFEDF shown in Figure 7.15 and 7.16. The greatest performance difference between PD, NPD and ANPD is observed when the system's total density reaches 9.0. DLAEDF with PD is capable of saving 62 % energy, which is 18 % and 39 % more than that of DLAEDF with NPD and ANPD, respectively. The very short deadlines close to or even shorter than the subtask's WCET, assigned by NPD or ANPD causes the processor to execute at the full speed. Running at full speed causes high power consumption. For ANPD, it is even worse if the task's actual execution time is greater than its average execution time. There might be missed local deadlines even the processor runs at full speed, which causes the speedup execution of the succeeding tasks. The performance of the three deadline assignment is close when the system is scheduled using DFEDF. DFEDF's sophisticated exploitation of system's slack overcomes the negative effect of very short local deadlines assigned by NPD and

ANPD.

7.4 Distributed DVS-EDF Scheduling Simulation

Simulations of system sof 10 processors demonstrate the performance differences between distributed DVS-EDF algorithms. The subtasks in the end-to-end task set are assigned to the processor according to MinΔP task assignment heuristic [42]. Proportional deadline assignment [22] is used to assign deadlines to subtasks. The unit communication energy cost, E_{Kbyte} is set to 0.01 mJ/B.

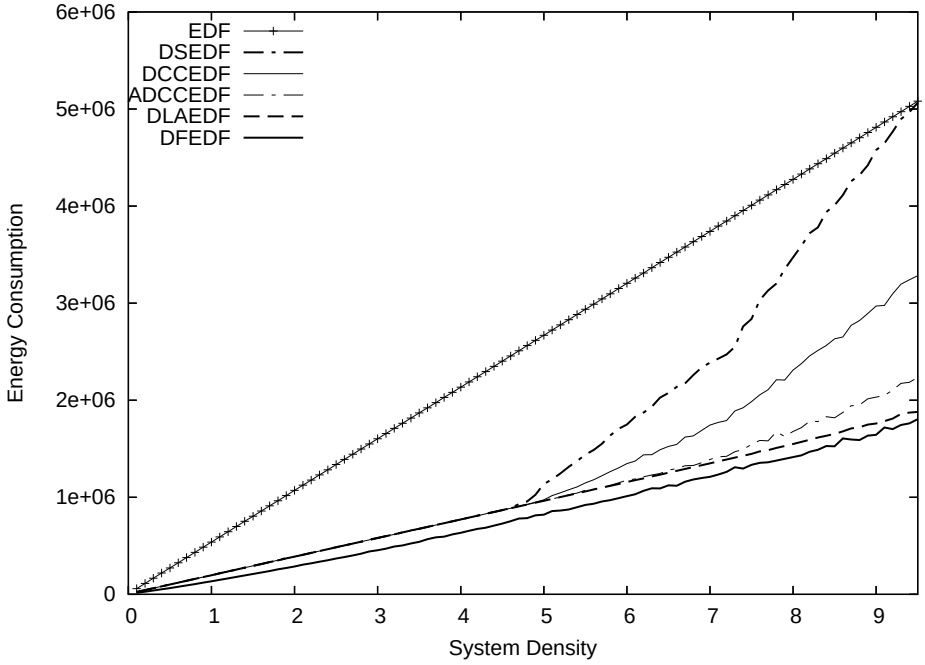
To compare the ability of each distributed DVS-EDF scheduling algorithm to exploit slack, two models of actual execution time (AET) are used in the simulation. The first set of simulations uses a task set with actual execution times equal to their WCET. The second simulates tasks with actual execution time of each job randomly generated between zero and the task's WCET with a modified Gaussian distribution. For the Gaussian-distributed actual execution times, the mean is one half the WCET and the standard deviation is 1 ms. Random values below 1 % of WCET are limited to 1 % of WCET, and random values greater than WCET are limited to the WCET.

Figure 7.17 compares the PROC1 and PowerPC power models with Gaussian distributed AET. The X-axis is the total system density of the distributed real-time system from 0.1 to 9.5. The Y-axis is the absolute system energy consumption with each distributed DVS-EDF scheduling algorithm.

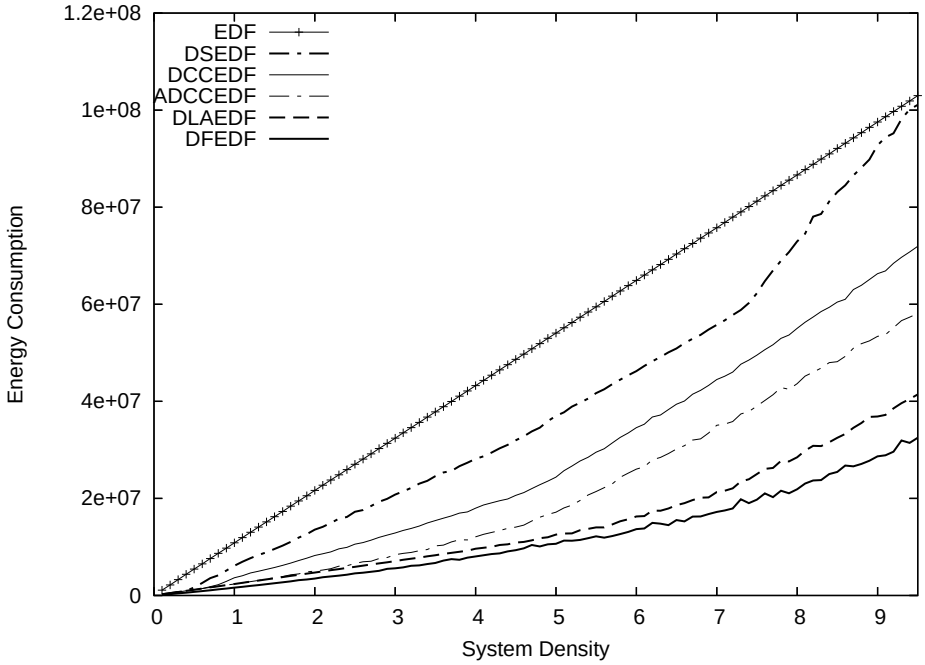
The energy consumption using any of the scheduling algorithms increases as the system's total density increases. The increase is linear with density for EDF, because the amount of processor idle time is decreasing linearly. For processor model PROC1 in Figure 7.17(a), When system density is below 4.5, all of the DVS scheduling algorithms can run at the slowest speed and thus use much less energy than EDF. Even at this low level of utilization, DFEDF is able to find enough more slack to noticeably reduce power consumption. Figure 7.17(b) shows the absolute energy consumption of each distributed DVS-EDF algorithms with PowerPC processor model. The more evenly distributed processor's operating points facilitates some effective DVS-EDF algorithms to saving more energy even when the system's density is low.

The relative performance difference of each algorithm is easier to see by comparing them relative to EDF. Figures 7.18 and 7.19 show energy normalized to the energy used by EDF on the Y-axis with PROC1 and PowerPC processor model, respectively.

Figure 7.18 shows that, except for DFEDF, energy consumption with all of the DVS algorithms maintains a constant ratio with EDF until density reaches 4.5.

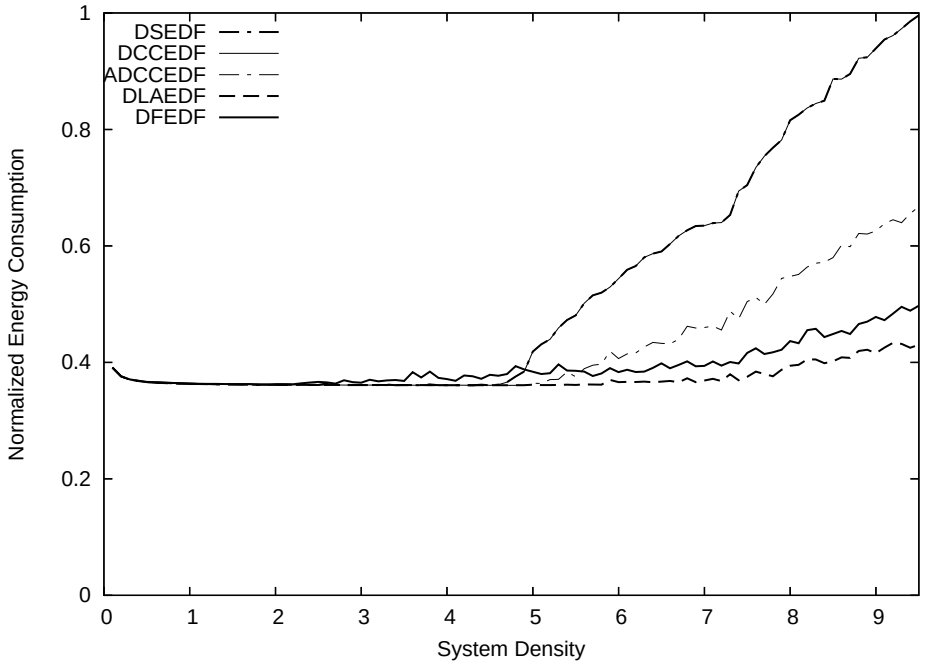


(a) Energy consumption with PROC1

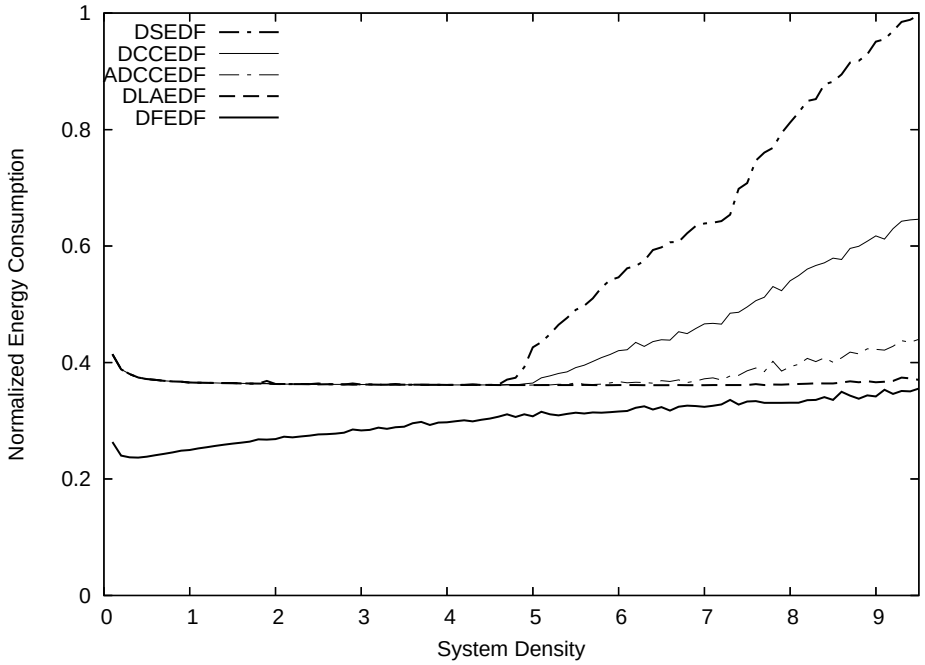


(b) Energy consumption with PowerPC

Figure 7.17: Absolute energy consumption with Gaussian distributed AET

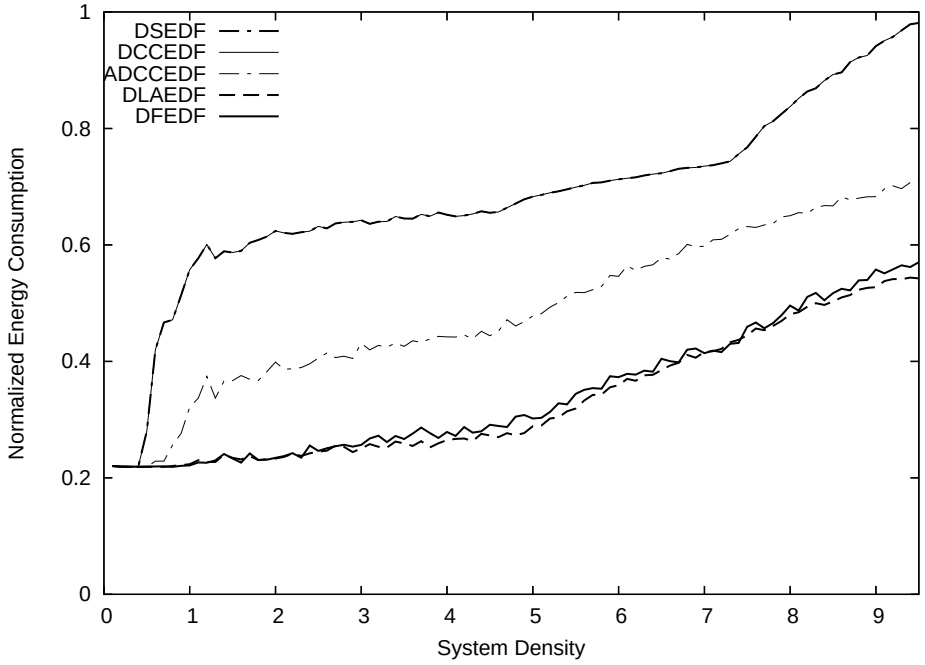


(a) System with AET = WCET

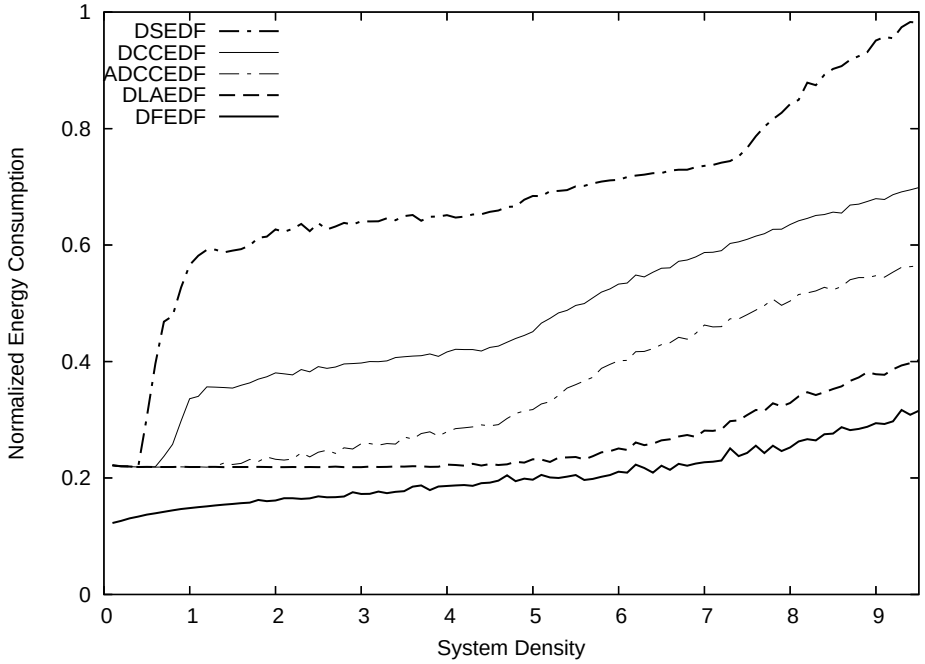


(b) System with Gaussian distributed AET

Figure 7.18: Energy consumption with PROC1



(a) System with AET = WCET



(b) System with Gaussian distributed AET

Figure 7.19: Energy consumption with PowerPC 405LP

DFEDF is able to take advantage of more slack than the other algorithms throughout the range of densities when actual execution time is less than worst case execution time.

DCCEDF shares the same performance with DSEDF in Figure 7.18(a), when the task's actual execution time takes the full WCET. However, Figure 7.18(b) shows that DCCEDF performs much better than SEDF when tasks have actual execution time less than their WCET. This performance difference is because DCCEDF is capable of exploiting dynamic slack as well as static slack in the system. The energy-consumption with DCCEDF increases faster than that with ADCCEDF when system density is greater than 5, because ADCCEDF is able to exploit more system slack than CCEDF. The tighter schedulability bound allows it to continue reducing processor speed at higher system densities. The DLAEDF and DFEDF perform better than the above scheduling algorithms using as little as 50 % and 45 % of energy used by EDF, respectively.

All distributed DVS-EDF scheduling algorithms except DSEDF have better performance when actual execution time is less than WCET. Since DSEDF sets the clock frequency once and never updates it, it cannot take advantage of slack made available from jobs that complete before their WCET. DCCEDF performs as much as 35 % better with Gaussian distributed AET. The ADCCEDF, DLAEDF and DFEDF in Figure 7.18(b) increase the performance by as much as 20 %, 8 %, and 14 % respectively when compared with Figure 7.18(a) .

The processor power model plays an even larger role in energy consumption than average execution time. Figure 7.19 shows the simulation results using the PowerPC 405LP power model. DSEDF has a large increase in the energy consumption when the system's density is about 0.4. This increase in energy consumption is caused by a large increase in supply voltage. According to Table 7.2, in order to speed up the processor by 80 %, from 100 MHz to 180 MHz it must increase its voltage by 4.17×, from 0.18 V to 0.75 V. Subsequent increases are more gradual starting around a density of 1.5. DLAEDF and DFEDF are both capable of more energy reduction when using PowerPC than that when using PROC1. As long as the system's density is below 6, they are able to take advantage of the PowerPC's 100 MHz operating point that consumes only 5.5 % maximum power. DLAEDF and DFEDF use as little as 21 % and 11 % of the energy used by EDF, respectively with the Gaussian-distributed average execution time.

SEDF computes the processor speed offline and schedules the tasks at the lowest constant speed that guarantees schedulability. Thus, DSEDF has no online overhead.

Though CCEDF and ADCCEDF update the instantaneous density at each scheduling point, CCEDF has a lower computation complexity of $O(1)$ than ADCCEDF. The computation of tighter bound of schedulability in ADCCEDF has a complexity of $O(n)$. The DLAEDF have a better performance than ADCCEDF with same computation complexity of $O(n)$ required by `defer` function. The DFEDF is the most complicated one among the DVS-EDF algorithms discussed in this paper. The computation complexity of online maximal schedule recomputation is determined by the number of tasks assigned on the processor and the ratio of the maximum period to the minimum period among all task's periods. When the period ratio is constant, the computation complex of DFEDF is $O(n^2)$.

7.5 Dynamic Task Set Admission Simulation

To measure the performance of our admission test and minimum deadline computation algorithm, we simulated adding tasks to a running uniprocessor DVS scheduler. For all simulations, the input task sets starts with independent regular periodic task set and one task is added at a time until utilization reaches 100 %. Each task is taken from one of three categories with equal probability. Short tasks have a period of 1 ms to 10 ms, medium tasks a period of 10 ms to 100 ms, and long tasks have periods from 100 ms to 1000 ms as described by Pillai and Shin [31]. The worst case execution time of each task is generated randomly based on its period and the system utilization. During the simulation, the actual execution time of each job is a randomly generated value between zero and its task's worst case execution time, with a uniform distribution. For all tests, deadlines are set to be equal to periods.

Simulations for our General Admission Test were done to measure the percentage of tasks that could not be admitted due to lack of slack that has been used by DVS in the schedule as a function of system utilization. The utilization of the task being added is fixed at 10%, and $D_{new,1}$ is set to D_{new}. The interval between adding two tasks is long enough to allow the system to reach a stable state. EDF and LAEDF DVS algorithms were tested to measure the performance. Each data point was run 100 times and the average is reported.

The result of the simulation shows that our proposed admission test admits new tasks with zero rejection rate in cases when the system's utilization is under 90%. The maximum rejection rate of 1% for EDF and 11% for LAEDF happens when the system's utilization reaches 100% after adding the new task. When scheduling new tasks with EDF, the new task should be schedulable as long as the total density is

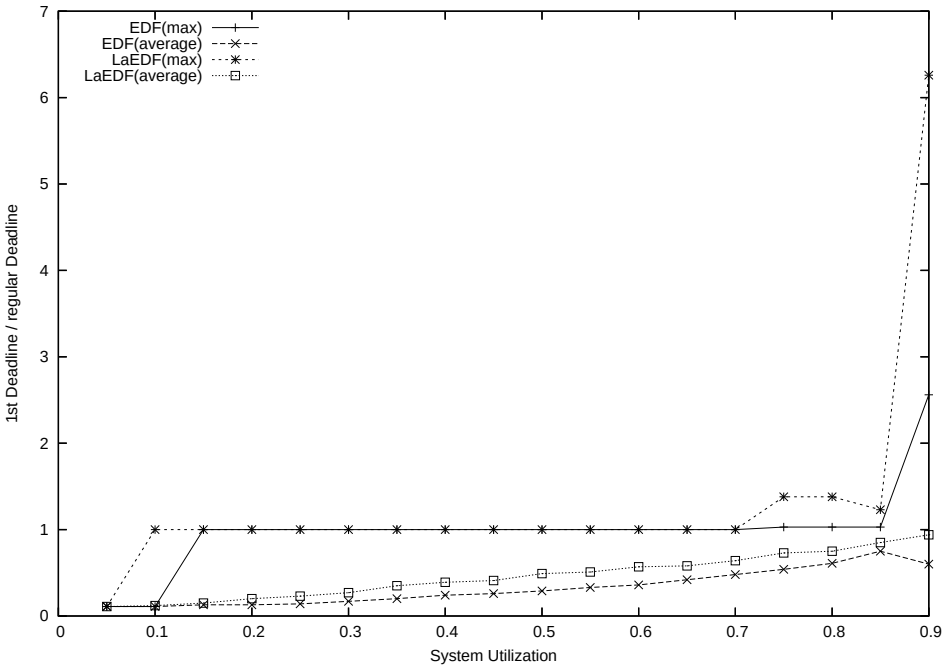


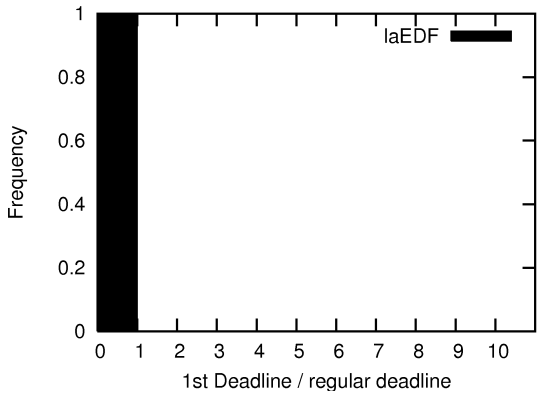
Figure 7.20: Computed deadline as a function of system utilization

no greater than one, though the total density may be higher for DVS algorithms. Although our admission test is not optimal, the maximum rejection rate of 1% for EDF is small. LAEDF is aggressive in exploring system's slack for slowing down job's execution. When the system's utilization is high, there is less system static slack time available for new job's execution. Thus, the chance of new task rejection increases.

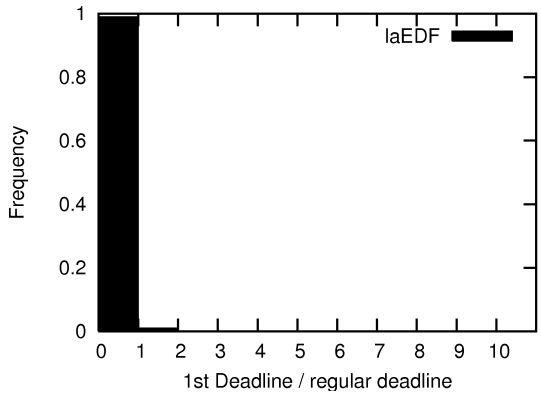
Figure 7.20 shows the performance of the minimum deadline computation algorithm. The minimum deadline computed using our algorithm is normalized to the new task's regular deadline. The utilization of the task being added to the system is 10%. Each pair of the average and maximum normalized deadline is generated by 100 simulation runs each with a different random seed. To obtain a better view of performance of our algorithm, the distribution of normalized deadlines for system utilizations 0.7, 0.8 and 0.9, is shown in Figure 7.21.

The simulation results show that the the maximum value of computed deadline by our algorithm for LAEDF is no greater than the regular deadline when system's original utilization is less than 70%, which means the new task's first job can be scheduled with the task's regular deadline. Larger deadline delay happens when the system's utilization increases. The maximum deadline delay ratio of 6.25 is observed when the system's utilization reaches 90% before adding the task with $u_i = 10\%$.

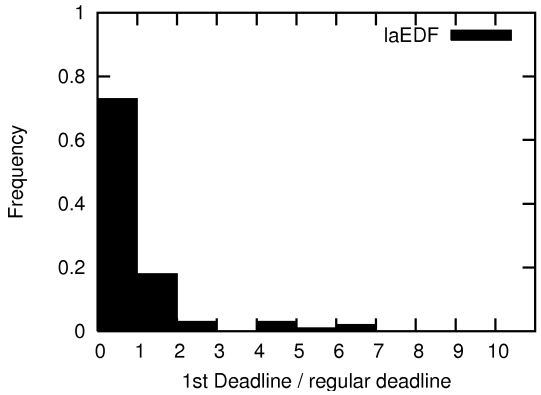
When utilization is 0.8, as in Figure 7.21 (b), 1% of deadlines are delayed. The



(a) system utilization = 0.7



(b) system utilization = 0.8



(c) system utilization = 0.9

Figure 7.21: Distribution of computed deadline

worst case delay is no greater than twice the length of new task's regular deadline. For real-time system that can tolerate small deadline delay of first job in the new task, this 1% new task is acceptable. When system's utilization is 90%, as shown in Figure 7.21 (c), there are 22% of deadlines take greater value than the regular deadline. The cause of the large deadline delay is the extreme lack of system slack. Among the 22% deadlines longer than the tasks' regular deadlines, there are 6% more than 2 times of regular deadline. If the system can tolerate a deadline delay of up to 2 times of task's regular deadline, our algorithm will only cause 6% of new task being rejected by the system.

Chapter 8: CONCLUSIONS AND FUTURE WORK

The dynamic voltage scaling for uniprocessor real-time system has been proved to successfully reducing the energy consumption. To save the energy consumption for distributed real-time system, the dissertation extended the uniprocessor DVS-EDF scheduling algorithms for partitioned distributed systems. Online task assignment and deadline assignment heuristics are studied to facilitate the energy-conserving performance of the extended DVS-EDF scheduling algorithms. The priority-driven scheduling can deal with dynamic task sets with tasks arriving and leaving the system. However, the simple addmission test for system scheduled without DVS does not stand for DVS scheduled real-time system. Two addmission algorithms are proposed in the dissertation enabling the DVS scheduled real-time system to accept arriving tasks when system is running. Simulations done to compare the performance of each algorithm discussed in the dissertation show that the distributed DVS-EDF scheduling algorithms can save up to 89 % of system energy consumption with MinΔP task assignment and PD deadline assignment. Among all task assignment and deadline assignment heuristics discussed in the dissertation, MinΔP task assignment and PD deadline assignment are capable of helping distributed DVS-EDF scheduling algorithm to save more energy. With the admission test proposed, the DVS-EDF scheduled system is able to accept over 80 % of arriving tasks when system is running.

The distributed DVS-EDF scheduling algorithms for partitioned distributed real-time system, such as DSEDF, DCCEDF, ADCCEDF, DLAEDF, and DFEDF, have all been seen to reduce energy consumption. The dissertation discussed extends the applicability of DVS-EDF algorithms to tasks with end-to-end precedence constraints or deadlines different than their periods. The changes to SEDF and CCEDF are as simple as substituting density for utilization. ADCCEDF modifies CCEDF to use a tighter schedulability bound, allowing slower processor speeds for the same workload. DLAEDF requires more extensive changes. Its deferable work computation must account for jobs whose deadline has passed, but whose next job in the task has not been released. FEDF requires the largest change. To overcome the release jitter in the partitioned real-time systems, it computes the available slack using a dynamically computed maximal schedule.

Simulations show the relative performance of the distributed DVS-EDF scheduling algorithms vary depending on power and task models. Regardless of processor model, all DVS algorithms except SEDF saved more energy when actual execution time was less than worst case execution time. The dynamic slack made available helps improve

the performance of those algorithms that can take advantage of it. In particular, DCCEDF, DLAEDF, and DFEDF improved in all cases when actual execution time is less than WCET.

A hypothetical processor power model based on a simple CMOS power model shows significant power savings with DVS. For DVS algorithms be able to take advantage of slack, the PowerPC model shows even higher savings, because of its extremely low power minimum speed mode. DFEDF has the best energy-conserving performance – using as little as 11 % of the energy used by EDF for the PowerPC. DLAEDF is next best, using as little as 21 % of the energy of EDF with a lower complexity than DFEDF. With dynamic workloads and an extremely low-power minimum-speed mode a well-designed DVS algorithm can reduce energy consumption by almost an order of magnitude.

How tasks are assigned to processors in partitioned distributed real-time systems affects the energy-conserving performance of DVS-EDF scheduling algorithms. Several task assignment heuristics have been studied in this dissertation. Two new task assignment algorithms, Communication-Aware Worst-Fit and MinΔP, are proposed. The Communication-Aware Worst-Fit tries to schedule tasks that communicate on the same processor to reduce overall communication cost. Tasks do not communicate or cannot be placed with their sibling tasks are assigned using Worst-Fit to balance the system workload. MinΔP uses estimated change in power to decide to which processor a task should be assigned.

When compared to existing well-known bin packing heuristics, Worst-Fit and Best-Fit, Communication-Aware Worst-Fit and MinΔP perform favorably. When SEDF scheduling is used, Communication-Aware Worst-Fit reduces power consumption due to communication and overall power in most cases, and MinΔP always performs better than Worst-Fit. When LAEDF or FEDF are used to schedule tasks, energy consumption was insensitive to task assignment. The ability of LAEDF and FEDF to use dynamic slack to stretch execution time allows them to keep processors running at low speeds most of the time, even when system density is high.

Among the four task assignment algorithms, the Best-Fit and Worst-Fit need the least information about the task set when making task assignment decisions – just the current processor and task densities. Communication-Aware Worst-Fit additionally needs to know each task's predecessor as well as the predecessor's processor. MinΔP requires not only the task information, but also power cost for processors and communication.

Further simulation results show that, except at low utilization, communication

cost adds little to overall energy consumption. Best-Fit and Communication-Aware Worst-Fit heuristics both avoid communication costs; Best-Fit by scheduling tasks on the most heavily loaded processor until no more tasks can be assigned to the processor, and Communication-Aware Worst-Fit by scheduling subtasks of the same end-to-end task on the same processor. However, unless the communication cost is relatively high, power due to communication is small compared to overall system power.

Overall, MinΔP provides the best power savings with SEDF. When using LAEDF or FEDF to schedule tasks, power consumption is nearly indistinguishable between algorithms. In this case other criteria, such as feasibility performance, are more important in selecting a task assignment algorithms. In particular, using Best-Fit only marginally increases power consumption, but has been shown to increase schedulability of tasks significantly.

Several different deadline assignment heuristics have been discussed in the dissertation, UD, ED, PD, NPD and ANPD. PD and NPD are expected to result in higher schedulability than other heuristics. NPD and the proposed ANPD are expected to have better energy-conserving performance than other deadline assignment approaches. When applied to the distributed real-time system with tasks frequently arriving or leaving the system, the overhead of each deadline assignment has to be considered. NPD and ANPD need global information of utilization on each processor, which requires higher online overhead than PD.

Simulation results show that the PD has the best energy-conserving performance than NPD and ANPD for the DLAEDF and DFEDF scheduled distributed real-time systems. DLAEDF is capable of saving 62 % of energy when using PD, which is 18 % and 39 % more than that using NPD and ANPD, respectively. In addition, local deadline misses are observed when assigning deadlines with NPD and ANPD. Although it is feasible as long as the end-to-end deadline is guaranteed [38], the local deadline missing degrade the performance of DVS-EDF scheduling algorithm.

The ordinary EDF admission tests are not sufficient when the system is scheduled by DVS-EDF scheduling algorithms, because the scheduler may defer too much work to allow a new task to be scheduled. An online admission tests and a deadline computation algorithm for adding periodic tasks to systems using real-time DVS scheduling is proposed and discussed in this dissertation. The first provides sufficient conditions for admission with any DVS algorithm capable of scheduling any task set that is schedulable by EDF, for example LAEDF [31]. The second algorithm computes a feasible deadline for the first job in the new task by which the new task

can be admitted by the system.

The admission test determines whether the first instance of a job can run before a given deadline, and the subsequent jobs in the task will be schedulable with any DVS algorithm that can schedule any set of tasks schedulable by EDF. Simulations show that when the maximum delay of the first job in the new task is equal to the task's relative deadline and LAEDF is used for scheduling, only 11% of tasks that can be admitted by EDF are rejected by our admission test. For our deadline computation algorithm, there are only 6% of computed deadlines task more than 2 times of regular deadline in the worst case.

The dissertation based the discussion mostly on the EDF scheduling, research on the energy-aware scheduling algorithms for fixed-priority scheduled partitioned distributed real-time system is one of the future work directions. Rate monotonic scheduling is a popular fixed-priority based scheduling algorithm used in industry. DVS has been applied to RM scheduled uniprocessor real-time systems, but little work has been done for distributed systems. Extension of uniprocessor DVS-RM to distributed real-time system would be profitable for large scaled real-time systems.

For the admission test and deadline computation algorithms presented in the dissertation for dynamic task set in the uniprocessor system, further work is needed to extend the algorithms to the distributed real-time systems with end-to-end tasks. Accepting the end-to-end task online involves the assignment of subtasks in the arriving end-to-end task. Although the online task assignment heuristics discussed in the dissertation could be used, the use of system's density or utilization in making assignment decision does not stand any more because of the DVS scheduling. By stretching the execution of tasks, DVS increases the instantaneous system density. An algorithm is needed to make the appropriate assignment decisions with low computational complexity.

BIBLIOGRAPHY

[1] J. H. Anderson and S. K. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems*, pages 428–435, March 2004.

[2] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time System Symposium*, page 95, December 2001.

[3] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*, page 113.2, April 2003.

[4] R. J. Baker, H. W. Li, and D. E. Boyce. *CMOS: Circuit Design, Layout, and Simulation*. IEEE Press, 1998.

[5] S. Baruah and J. Anderson. Energy-aware implementation of hard-real-time systems upon multiprocessor platforms. In *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems*, pages 430–435, August 2003.

[6] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations (2000). In *Proceedings of 27th International Symposium on Computer Architecture*, pages 83–94, June 2002.

[8] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE transactions on computers, 1995*, pages 1429–1442, December 1995.

[9] A. Burchard, Y. Oh, J. Liebeherr, and S. Son. A linear-time online task assignment scheme for multiprocessorsystems. In *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 28–31, Seattle, WA, USA, May 1994. IEEE.

[10] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[11] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems*, pages 191–199. IEEE, June 1997.

[12] U. M. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 23–30. IEEE Computer Society, July 2003.

[13] C. D.Roychowdlhury, I.Koren and Y.-H. Lee. A voltage scheduling heuristic for real-time task graphs. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 741–750, June 2003.

[14] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback EDF scheduling for embedded systems with real-time constarints. In *ACM SIGPLAN Joint Conference LCTES'02*, pages 213–222, June 2002.

[15] J. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. *Approximation algorithms for NP-hard problems*, pages 46–93, 1996.

[16] G. W. Greenwood, C. Lang, and S. Hurley. Scheduling tasks in real-time systems using evolutionary strategies. In *WPDRTS '95: Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, page 195, Washington, DC, USA, 1995. IEEE Computer Society.

[17] F. Gruian. System-level design methodes for low-energy architectures containning variable voltage processors. In *Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 1–12. Springer-Verlag, 2000.

[18] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and DVS processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 46–51. ACM Press, 2001.

[19] P.-E. Hladik, H. Cambazard, A.-M. Dplanche, and N. Jussien. How to solve allocation problems with constraint programming. In I. Puaut, editor, *Work In Progress of the 17th Euromicro conference on real time systems (ECRTS'05)*, pages 25–28, Palma de Mallorca, Spain, 2005. IRISA, Rennes, France.

[20] C. Im and S. Ha. Dynamic voltage scaling for real-time multi-task scheduling using buffers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 88–94. ACM Press, 2004.

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

[22] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

[23] J. Lopez, J. Diaz, M. Garcia, and D. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Workshop on Real-Time Systems, 2000*, pages 25–33, June 2000.

[24] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 156–163. ACM Press, 2002.

[25] J. Luo and N. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proceedings of 2000 International Conference on Computer-Aided Design*, pages 357–364, November 2000.

[26] J. Luo and N. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proceedings of the 7th Asia and South Pacific and the 15th International Conference on VLSI Design*, pages 719–726, Bangalore, India, January 2002. IEEE.

[27] D. Marinov, D. Magdic, A. Milenkovic, J. Protic, I. Tartalja, and V. Milutinovic. An approach to characterization of parallel applications for dsm systems. In *Proceedings of the 31st HICSS, IEEE Computer Society Press*, pages 782–784, January 1998.

[28] M. A. Moncus, A. Arenas, and J. Labarta. Energy aware edf scheduling in distributed hard real time systems. In *RTSS Work-in-Progress*, pages 103–106, December 2003.

[29] R. Nossal. An evolutionary approach to multiprocessor scheduling of dependent tasks. *Parallel and Distributed Processing*, pages 279–287, 1998.

[30] K. Nowka, G. Carpenter, E. M. Donald, H. Ngo, B. Brock, K. Ishii, T. Nguyen, and J. Burns. A 0.9 v to 1.95 v dynamic voltage-scalable and frequency-scalable32 b powerpc processor. Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International, February 2002.

[31] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 89–102, October 2001.

[32] G. Quan and X. S. Hu. Minimal energy fixed-priority scheduling for variable voltage processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1062–1071, August 2003.

[33] N. R. Ramesh Mishra and D. Zhu. Energy aware scheduling for distributed real-time systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[34] P. Rong and M. Pedram. Power-aware scheduling and dynamic voltage setting for tasks running on a hard real-time systems. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pages 473–478, Yokohama, Japan, 2006. ACM.

[35] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the simos approach. *Parallel and Distributed Technology: Systems and Applications, IEEE*, pages 34–43, Winter 1995.

[36] C. Rusu, R. Melhem, and D. Mossé. Maximizing the system value while satisfying time and energy constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 1–10. IEEE, 2002.

[37] M. T. Schmitz and B. M. Al-Hashimi. Considering power variations of dvs processing elements for energy minimisation in distributed systems. In *Proceedings of the 14th International Symposium on Systems Synthesis*, pages 250–255, Montréal, Québec, Canada, October 2001.

[38] J. Sun. *Fixed priority scheduling of end-to-end periodic tasks*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[39] A. K. Tripathi, B. K. Sarker, and N. Kumar. A ga based multiple task allocation considering load. *International Journal of High Speed Computing*, pages 203–214, June 2000.

[40] C. Wang and W. R. Dieter. DVS algorithms for systems with relative deadlines shorter than their periods. Technical Report TR-ECE-2005-12-07-01, University of Kentucky, Electrical and Computer Engineering Dept., 453 F. Paul Anderson Tower, Lexington, KY 40506-0046, `http://www.engr.uky.edu/ dieter/pub/TR-ECE-2005-12-07-01.pdf`, December 2005.

[41] C. Wang and W. R. Dieter. Dynamic voltage scaling for priority-driven distributed real-rime systems. Technical Report ECE=2007-10-19, University of Kentucky, 453 F. Paul Anderson Tower, Lexington, KY 40506-0046, October 2007.

[42] C. Wang and W. R. Dieter. Power-aware task assignment for priority-driven distribtued real-time system. Technical Report ECE-2007-10-17, University of Kentucky, 453 F. Paul Anderson Tower, Lexington, KY 40506-0046, October 2007.

[43] K. C. Y. Shin and T. Sakurai. Power optimization of real-time embedded system on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 365–368, November 2000.

[44] W. Ye, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Design Automation Conference*, pages 340–345, 2000.

[45] Y. Yu and V. Prasanna. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *Proceedings of Ninth International Conference on Parallel and Distributed Systems, 2002*, pages 341–348, December 2002.

[46] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, December 2001.

[47] Y. Zhu and F. Mueller. Preemption handling and scalability of feedback DVS-EDF. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2002.

[48] Y. Zhu and F. Mueller. Feedback dynamic voltage scaling DVS-EDF scheduling: Correctness and PID-feedback. Technical Report TR-2003-13, North Carolina State University, June 2003.

[49] Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 203–212, Chicago, IL, USA, June 2005. ACM.

[50] Y. Zhu and F. Mueller. Feedback EDF scheduling of real-time tasks exploiting dynamic voltage scaling. *Real-Time Systems Journal*, 31(1–3):33–63, December 2005.

VITA

Chenxing Wang was born in Sichuan, People's Republic of China on November 3, 1975.

She attended Hunan University in September 1994 and graduated with a B.S. degree in Electrical Engineering in July 1998. She was awarded Hunan University Scholarship for Outstanding Academic Performance in 1995, 1996 and 1997.

From September 1998 to May 2001, she studied for her Master degree in Electrical Engineering at the University of Electronic Science and Technology of China.

After graduated with her Master of Science degree, she joined Maipu communication company in China as a firmware engineer in June, 2001.

In January, 2002, she attended the Ph.D. program in University of Kentucky. She started the research on power-aware scheduling for distributed real-time embedded system in 2004. During her graduate study, she was awarded Kentucky Opportunity Fellowship in 2005 and 2006.