ENERGY-AWARE SCHEDULING FOR STREAMING APPLICATIONS

by

Ruibin Xu

B.E., Guangdong University of Technology, P.R.China, 1996

M.S., Zhongshan University, P.R.China, 1999

Submitted to the Graduate Faculty of

the Arts and Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Ruibin Xu

It was defended on

January 4th 2010

and approved by

Dr. Rami Melhem

Dr. Daniel Mossé

Dr. Bruce Childers

Dr. Jun Yang

Dissertation Advisors: Dr. Rami Melhem,

Dr. Daniel Mossé

ENERGY-AWARE SCHEDULING FOR STREAMING APPLICATIONS

Ruibin Xu, PhD

University of Pittsburgh, 2010

Streaming applications have become increasingly important and widespread, with application domains ranging from embedded devices to server systems. Traditionally, researchers have been focusing on improving the performance of streaming applications to achieve high throughput and low response time. However, increasingly more attention is being shifted to power/performance trade-off because power consumption has become a limiting factor on system design as integrated circuits enter the realm of nanometer technology.

This work addresses the problem of scheduling a streaming application (represented by a task graph) with the goal of minimizing its energy consumption while satisfying its two quality of service (QoS) requirements, namely, throughput and response time. The available power management mechanisms are dynamic voltage scaling (DVS), which has been shown to be effective in reducing dynamic power consumption, and vary-on/vary-off, which turns processors on and off to save static power consumption.

Scheduling algorithms are proposed for different computing platforms (uniprocessor and multiprocessor systems), different characteristics of workload (deterministic and stochastic workload), and different types of task graphs (singleton and general task graphs). Both continuous and discrete processor power models are considered. The highlights are a unified approach for obtaining optimal (or provably close to optimal) uniprocessor DVS schemes for various DVS strategies and a novel multiprocessor scheduling algorithm that exploits the difference between the two QoS requirements to perform processor allocation, task mapping, and task speed scheduling simultaneously.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGEMENT

First and foremost, I would like to thank my advisers, Dr. Rami Melhem and Dr. Daniel Mossé, for their guidance, support, and help throughout my Ph.D. study. They were always there to listen and give advice. They taught me how to do scientific research and tackle hard problems. No matter what career path I take, industry or academia, they are forever my role models.

A special thanks to Dr. Bruce Childers, who always asked good questions about my work, which in turn led to improvement of my work. I would also like to thank Dr. Jun Yang for agreeing to serve on my Ph.D. committee and for providing helpful comments on my work.

I am grateful to all members of the collaboration in the Power-Aware Real-Time Systems (PARTS) research group. Specifically, I want to thank Cosmin Rusu, Dakai Zhu, Navine AbouGhazaleh, Matt Craven, and Alexandre Ferreira for their kind help. I really enjoyed working with these wonderful people.

On the personal level, I would like to thank my wife, Yuanbing Du, for her constant love, encouragement, and support during my educational career. Without her, I would have been a very different person and would not have finished my Ph.D. study. Last but not least, I would like to dedicate this work to my mother, Jinlan Huang, who passed away just before I came to the US to pursue my Ph.D.. She had always been telling me that having higher education would be my only way out. She was so right. I believe my Ph.D. graduation would have made her the happiest person in the world.

1.0 INTRODUCTION

Streaming applications are those that operate on continuous streams of data. They have become increasingly more important and widespread. Examples include Internet audio and video streaming. These streaming media applications have already been consuming a significant portion of the Internet bandwidth and their use continues to grow. Other examples of streaming applications include automatic target recognition (ATR) found in radar systems [43], which does pattern matching of targets in images that are continuously fed from sensors.

Traditionally, researchers have focused on improving the performance of streaming applications. There are two typical performance metrics for streaming applications: *throughput* and *response time*. The data stream that a streaming application operates on can be abstracted as a series of *requests* (e.g., a frame in video streaming is a request) and the streaming application is servicing the requests (e.g., decoding frames in video streaming) in succession. Thus, throughput is defined as the number of requests that a streaming application services in one second and response time is defined as the maximum time allowed to service one request. Clearly, high performance for a streaming application is synonymous with high throughout and low response time.

In general, there are two approaches to improving the performance of streaming applications: a hardware approach and a software approach. The hardware approach refers to designing faster computer systems while the software approach is through developing smarter implementation and/or scheduling algorithms. Over the past three decades, the hardware approach has overshadowed the software approach due to Moore's Law. Processor frequencies have gone from 4K Hz to 4G Hz. As a result, the performance of streaming applications has improved dramatically. However, the price paid is that of drastically increased power consumption. The increased power consumption results in increased energy consumption,

which is especially detrimental for battery-powered systems, and generates excessive amount of heat that calls for expensive sophisticated packaging and cooling techniques. The generated heat, if not effectively removed, can also reduce system reliability. Thus, there is a fundamental trade-off between performance and energy. Since power/energy consumption can no longer be ignored in modern systems, increasingly more research attention has been shifted from just performance to energy-performance trade-off, which is the focus of this dissertation. Because streaming applications are continuous in nature and usually compute-intensive, most of them are power-demanding and energy-hungry. Thus, there is a great need to optimize energy consumption for streaming applications, while satisfying two typical quality-of-service (QoS) requirements, namely, throughput and response time.

Modern systems usually provide two power management mechanisms at the operating system level. The first one is *vary-on/vary-off* (or on-off for short), which refers to turning off or putting processors into sleep mode to save static power consumption resulting from leakage current. The second one is *dynamic voltage scaling* (DVS), which involves dynamically adjusting the voltage and frequency/speed[1] of a processor to reduce dynamic power consumption resulting from switching activities in circuits. When dealing with multiprocessor systems, another fundamental trade-off between static power and dynamic power consumption enters the picture. Assuming perfect parallelism, executing a streaming application on more processors increases the static power consumption of a multiprocessor system, while decreasing its dynamic power consumption. Hence, when running a streaming application in multiprocessor systems, it is necessary to combine these two power management mechanisms to optimize the energy consumption of the streaming application.

This dissertation assumes a given computer system (either uniprocessor or multiprocessor) that provides the aforementioned two power management mechanisms and a streaming application that is represented by a task graph. The goal is to design *scheduling algorithms* to schedule this streaming application to execute on this computer system (i.e., the software approach) to minimize the energy consumption of the streaming application while satisfying two QoS requirements, *throughput* and *response* time. The scheduling outcome includes the number of processors assigned to execute the streaming application, the task to processor

[1]In this dissertation, frequency and speed are used interchangeably.

mapping, and the execution speed of each task. A scheduling algorithm for uniprocessor systems is also called *DVS scheme* since DVS is the main power management mechanism. Note that in the context of real-time systems, the streaming application is equivalent to a periodic application, where the response time requirement is the deadline and the reciprocal of throughput is the period.

The following requirements should be satisfied in the scheduling algorithms: (i) *effectiveness*: the algorithms need to achieve good performance in terms of energy savings; (ii) *efficiency*: the offline computation of the scheduling algorithms must be tractable (i.e., at most polynomial time) and the online computation of the scheduling algorithms must be done in no more than linear time to minimize the scheduling overhead; (iii) *practicality*: the scheduling algorithms must take into consideration practical issues such as limited number of discrete speeds available in a processor and speed change overhead, which may not be ignored in practice.

Bearing the above requirements in mind, a comprehensive treatment on energy-aware scheduling for streaming applications is given. Different underlying computing platforms (uniprocessor and multiprocessor systems), different characteristics of the workload (deterministic and stochastic workload), and different types of task graphs (single task and general task graphs) are taken into account. Furthermore, two processor power models are considered. The first one is an ideal model in which the processor speed can be tuned continuously and unrestrictedly, and the speed change overhead is ignored. The second one is a realistic model, in which the processor has a limited number of discrete speeds and the speed change overhead is considered. The ideal model is of great simplicity and makes it possible to derive elegant and optimal scheduling algorithms through which insight into the problems under investigation is gained. However, the *ultimate goal* is to obtain scheduling algorithms under the realistic model.

The contributions of this doctoral work are as follows.

1. For uniprocessor scheduling on stochastic workload under the ideal processor model, an optimal inter-task DVS (for multiple tasks and speed can be changed only at task boundaries) scheme is proposed and a unified view of the optimal intra-task DVS (for a single task and speed can be changed inside a task), inter-task DVS, and hybrid DVS

(for multiple tasks and speed can be changed inside a task) schemes is provided [65, 67].

2. For uniprocessor scheduling on stochastic workload under the realistic processor model
   - A new intra-task DVS scheme called PPACE (Practical Processor Acceleration to Conserve Energy) is proposed. PPACE is a fully polynomial time approximation scheme (FPTAS) that can give performance guarantees and achieve energy savings very close to the optimal solution [68].
   - A new inter-task DVS scheme called PITDVS2 (Practical Inter-Task DVS using 2 speeds) is proposed and experimental results show that it outperforms the existing DVS schemes. It is also showed that simple extensions to optimal DVS schemes obtained under the ideal processor model do not necessarily generate DVS schemes that perform well in practice [67].
   - A unified practical approach for obtaining optimal (and provably close to optimal) stochastic inter-task, intra-task, and hybrid scheduling algorithms is proposed. The approach is based on a function approximation technique that also falls in the category of fully polynomial time approximation schemes. As a result, tight upper bounds on energy savings for stochastic DVS schemes is established and this approach can be used to evaluate existing DVS schemes [65].
3. For multiprocessor scheduling of a single task, a simple master-slave scheme that executes different instances of the task on multiple processors is proposed. Based on this scheme, scheduling algorithms for deterministic and stochastic workload, and for the ideal and realistic processor model are devised.
4. For multiprocessor scheduling of a task graph under the realistic processor model
   - A novel scheduling algorithm called Scheduling2D for deterministic workload is proposed. This algorithm exploits the difference between the two QoS requirements and performs processor allocation, task mapping, and task speed scheduling simultaneously. The design of this algorithm shows that the static power of processors has an important impact on the scheduling of streaming applications and high static power could lead to servicing requests faster than the response time requirement in order to save energy. Experimental results show that Scheduling2D achieves significant

energy savings over existing scheduling algorithms that only consider the response time requirements [66].

- The Scheduling2D algorithm is further extended to the SScheduling2D algorithm to deal with stochastic workload.

To the best of my knowledge, this dissertation is the first work to address energy-aware scheduling on multiprocessor systems considering both throughput and response time constraints.

This dissertation is organized as follows. First, the background and related work for this dissertation are described in Chapter 2. The models, problem description, and evaluation methodology are presented in Chapter 3. Chapters 4 and 5 present scheduling algorithms and evaluation results for uniprocessor and multiprocessor systems, respectively. Chapter 6 concludes this dissertation and Chapter 7 elaborates future research to extend this work.

2.0 BACKGROUND AND RELATED WORK

2.1 STREAMING APPLICATIONS

Continuous processing on a stream of data is the main characteristic of streaming applications. Stephens et al. [57] gave an excellent survey of stream processing. Thies et al. [58] pointed out a number of important properties of streaming applications. There have been a number of programming languages that are related to stream processing. In general, task graphs can be extracted from programs written in these languages.

The latest language designed for stream processing is MIT Streamit [58]. In Streamit, programs are represented as graphs, where nodes represent computation and edges represent FIFO-ordered communication of data. The basic programmable unit is a filter and filters can be composed into stream graphs by using three hierarchical structures: pipeline, split-join, and feedback loops.

Previous works on scheduling streaming applications for multiprocessor systems focused on performance issues. Bokhari et al. [12] gave an optimal algorithm for mapping a chain of tasks to multiprocessor systems. In [26], a stream compiler for Streamit is presented for scheduling streaming applications on communication-exposed architectures (e.g., the MIT RAW). Agarwalla et al. [4] developed a scheduler called Streamline to schedule streaming applications on Grid systems. Various heuristics [50, 55, 49] were proposed to use parallel processing and pipelining to maximize throughput or minimize the number of processors in synthesizing task graphs for multiprocessor systems. However, these works did not take energy consumption into consideration and this dissertation fills this void.

Streaming applications can be modeled as real-time applications, as described next.

2.2 REAL-TIME SYSTEMS

In real-time systems, the correctness of an application depends not only on the correctness of its output, but also the time it takes to produce its output. A real-time application can only start execution after its *release time* (defined to be the instant of time at which the application becomes available for execution) and must finish the execution correctly before its *deadline* (defined to be the instant of time by which the application is required to be completed). The response time of a real-time application is the length of time from its release time to the instant when it finishes. A real-time system can be either *hard real-time* or *soft real-time*. The former means that the deadline is a strict requirement and deadline messes may result in system failure, while the latter means that occasional deadline misses are tolerable and the performance requirement is a statistical one.

A real-time application generally consists of a set of tasks, which, together with the precedence constraints, constitute a task graph. Each task has a worst-case execution time (WCET), which can be obtained through profiling or analysis. In the context of energy-aware real-time scheduling, the task workload is often characterized by worst-case execution cycles (WCEC). The traditional WCET can be computed by dividing WCEC by the maximum frequency of the processor. Streaming applications that this dissertation deals with are real-time applications because they have response time requirement.

The most common task model in real-time systems is the periodic task model. In this model, each task is executed repeatedly at regular time intervals in order to provide a function of the system on a continuing basis. The time between consecutive release times of a task is called the *period* of the task. Deadlines are generally relative to the beginning of periods. If all tasks share a common period and an identical first release time, we call this type of real-time systems *frame-based systems*. Each period is called a *frame*. When tasks in an application have precedence constraints, a partial order on the execution of tasks is imposed. In addition, tasks can be *preemptive* or *non-preemptive*. While the execution of a preemptive task can be interrupted by another task and resume later, a non-preemptive task cannot be interrupted before its completion.

A streaming application can be modeled as tasks executing in a frame-based real-time

system. The throughput, defined as the number of requests per second, is equal to the reciprocal of the frame length. The response time requirement, defined as the maximum time allowed to service a single request, is equivalent to the task deadline.

The scheduling in real-time systems is to decide which task is executed on which processor at what time. For energy-aware scheduling, the speed at which a task is executed also needs to be determined. A schedule is said to be feasible if the precedence constraints, timing constraints, as well as any other constraints are satisfied.

2.3 POWER MANAGEMENT MECHANISMS

Power consumption in a CMOS processor can be divided into three parts: dynamic, static, and short-circuit power [44]. Short-circuit power is only consumed during signal transitions and is generally negligible [44]. Dynamic power is due to switching activities in the circuit and is roughly proportional to the input voltage squared times the operating frequency [60]. But input voltage and operating frequency are not independent and there is a minimum voltage required to drive the circuit at the desired frequency. The minimum voltage is approximately proportional to the frequency, leading to the conclusion that the power is proportional to the frequency cubed [14]. Since the time taken to run a program is inversely proportional to the operating frequency, quadratic energy savings can be achieved at the expense of just linear performance loss through DVS [30].

Static power consumption stems from leakage current that exists even in the absence of switching activities in a circuit. In traditional CMOS circuits, static power can be ignored because dynamic power dominates. However, this is not true any more considering the trends of CMOS circuit technologies. A five-fold increase in the leakage power is estimated with each technology generation [13]. Thus, power management schemes must take static power into consideration. Turning a processor off or putting it in sleep mode are general mechanisms to save static power.

2.4 ENERGY-AWARE UNIPROCESSOR SCHEDULING

In energy-aware uniprocessor scheduling, DVS is the main power management mechanism to be considered. This is because whether to turn off the only processor in the system is straightforward, that is, we turn off the processor when it has nothing to run. Thus, we also call the scheduling algorithms for uniprocessor systems *DVS schemes*. There are three types of DVS schemes: inter-task, intra-task, and hybrid [34]. Inter-task DVS schemes focus on allotting system time to multiple tasks and schedule speed changes only at each task boundary (i.e., the execution speed for a task is constant for each instance executed), while intra-task DVS schemes focus on how to schedule speed changes within a single task instance given an allotted amount of time. Hybrid DVS schemes are combination of intra-task and inter-task DVS schemes. There has been a large amount of work done in real-time systems related to DVS schemes. Next, we will review the related work for each type of DVS schemes.

2.4.1 Inter-task DVS Schemes

Inter-task DVS schemes differ in the way slack is allotted to tasks in the system. Slack includes *static slack* due to system under-utilization when each task is assumed to run for its worst-case execution cycles (WCEC) and *dynamic slack* due to early completion of tasks. The concept of *speculative speed reduction* was introduced in [48], which proposed three DVS schemes (i.e., Greedy, Proportional, and Statistical schemes) with different speed reduction aggressiveness for frame-based real-time systems. The Proportional scheme distributes the slack proportionally among all unexecuted tasks, while the Greedy scheme is much more aggressive and gives all the slack to the next ready-to-run task. The Statistical scheme uses the average-case execution cycles (ACEC) of tasks to predict the future slack. Many existing DVS schemes proposed for real-time systems can be classified as Proportional or Greedy. The cycle-conserving scheme and the look-ahead scheme for periodic real-time systems with variable workloads were proposed in [51]. When used in frame-based systems, the cycle-conserving scheme is equivalent to the Proportional scheme, while the look-ahead scheme is equivalent to the Greedy scheme. The scheme for fixed-priority real-time systems

proposed in [56], when used in frame-based systems, is equivalent to the Greedy scheme.

To be able to navigate the full spectrum of speculative speed reduction, Aydin et al. proposed a DVS scheme in which system designers can set a parameter to control the degree of speed reduction aggressiveness [8]. In fact, the optimal speed reduction aggressiveness depends on the variability of the workloads, as will be shown in Section 4.2.2. The Statistical scheme attempts to capture the variability of the workloads by using the average-case execution cycles (ACEC) of each task, which does not contain sufficient probabilistic information. Our inter-task DVS schemes in Section 4.2.2 automatically choose the degree of speed reduction aggressiveness to minimize the expected energy consumption, based on the probability distribution of the tasks' execution cycles.

2.4.2 Intra-task DVS Schemes

For a single task and a given deadline, Lorch et al. showed that if a task's computational requirement is only known probabilistically, there is no constant optimal speed for the task and the expected energy consumption is minimized by gradually increasing the speed as the task progresses [40, 42]. They call this approach PACE (*Processor Acceleration to Conserve Energy*). DVS schemes similar to PACE have also been proposed in [71, 27]. They differ in the way of patching the speed schedule obtained under an ideal processor model (i.e., the speed can be adjusted continuously and there is no speed change overhead) to fit a realistic processor model (i.e., considering discrete speed and speed change overhead). Among these intra-task DVS schemes, PACE and GRACE (*Global Resource Adaptation through CoopEration*) [71] can be used for soft real-time systems as they have a parameter to control the percentage of deadlines to be met. When this parameter is set to 100%, they are targeted at hard real-time systems.

2.4.3 Hybrid DVS Schemes

A hybrid DVS scheme can be obtained by using an inter-task DVS scheme as the basis and plugging in an intra-task DVS scheme [34, 71]. However, such hybrid schemes are inherently suboptimal because they ignore the interaction between inter-task and intra-task DVS. The

optimal hybrid DVS scheme under the ideal processor model extends PACE to multiple tasks in frame-based real-time systems [72]. However, the authors of [72] did not provide any solution to patch their scheme to be used in practice.

Another interesting result, applicable to both hybrid and inter-task DVS, is that different ordering of tasks in real-time systems results in different energy consumption [28, 29], which led to a number of heuristics to obtain a "good" ordering of tasks. This is complementary to our work because we focus on finding speed schedules given the ordering of tasks.

2.5 ENERGY-AWARE MULTIPROCESSOR SCHEDULING

Multiprocessor systems are quickly becoming the dominant computer platform as more and more chip multiprocessors (CMPs) available in the market. By combining multiple small processor cores on a single chip, CMPs continue to push the processor performance growth beyond the clock rate limit. Several chip makers have already released CMPs, such as IBM/Sony/Toshiba's 9-core CELL [1] and the 80-core prototype by Intel [2]. The trend is that more and more processor cores will be seen on a single computer system.

There are four key elements in energy-aware multiprocessor scheduling for streaming applications: (i) streaming applications are represented by task graphs; (ii) multiprocessors; (iii) quality of service constraints (i.e., throughput and response time); (iv) energy-aware scheduling using on-off and DVS. We will review related work that contains these elements.

Much research has been done on energy-aware scheduling of multiple independent tasks for multiprocessor systems using DVS (e.g., [62, 38, 16]). This dissertation focuses on scheduling of task graphs. There is also a lot work on energy-aware scheduling of task graphs using DVS and assuming equal period and deadline. In [23, 61], scheduling algorithms for simple special types of task graphs were proposed. For general task graphs, Mishra et al. proposed several heuristics to obtain the execution speed of each task assuming that the number of processors and task mapping are given [45]. Andrei et al. proposed a convex programming based approach, again, assuming that the number of processors and task mapping are given [5]. Cong et al. proposed a priority based heuristic to perform task mapping and a mathe-

matical program based approach to perform speed scheduling [18]. All these algorithms can be used as a component in our scheduling algorithm for general task graphs in Section 5.3.2. By simply trying every possible number of processors [37], all these work can be extended to consider on-off. In Section 5.3.2, we propose a better approach based on hill-climbing.

Before DVS emerged as an important power management mechanism, various approaches [50, 55, 49, 10] were proposed to use parallel processing and pipelining to maximize throughput or minimize the number of processors in scheduling task graphs on multiprocessor systems. Because energy consumption was not under consideration, straightforward application of these approaches cannot fulfill the need for energy optimization. On one hand, scheduling approaches that maximize throughput tend to use more processors, which is not energy efficient for high static power, and do not guarantee to comply with the response time requirement. On the other hand, scheduling approaches that minimize the number of processors tend to use fewer processors, which is not good for high dynamic power. Our scheduling algorithms in Section 5.3 also apply parallel processing and pipelining. However, we use them as *energy reduction techniques* and focus on finding appropriate number of processors to embrace the trade-off between static and dynamic power, and allotting appropriate amount of time to each task to stretch its execution.

Combining on-off and DVS to exploit the trade-off between static and dynamic power consumption has been used in multiprocessor-like settings by a number of researchers. In [22], Elnozahy et al. proposed a power management policy to determine the optimal number of online servers and corresponding operating frequency to minimize the energy consumption of clusters. In [69], Xu et al. tackled a similar problem considering several practical issues. In [9], Anderson et al. studied energy-efficient synthesis of periodic task systems on multiprocessor platforms. However, all of the above research dealt with independent tasks and considered only deadline constraint. Thus, they cannot be applied straightforwardly to task graph scheduling and multiple-constraint scheduling.

Kim et al. [33] explored the effectiveness of the simultaneous application of pipelining and parallel processing as a total power reduction technique in uniprocessor design. Our work in Section 5.3 is different from theirs in several aspects: (i) They focused on uniprocessor that is under a single voltage domain while we focus on multiprocessor systems and processors

can operate at different voltages and frequencies and have the capability of turning on/off; (ii) They assumed idealized unlimited parallelism in instruction streams while we focus on task graph mapping; (iii) They only considered throughput constraint while we consider two QoS requirements; (iv) Their parallel processing width (instruction issue width) is fixed for all stages while we can use different number of processors at different stages. (v) Their goal was to minimize the power consumption while we are trying to minimize the energy consumption.

3.0 MODELS AND PROBLEM DESCRIPTION

3.1 APPLICATION MODEL

A streaming application is modeled as a task graph $G(V, E)$, which is a directed acyclic graph (DAG), in order to exploit parallelism inside the application. The vertex $v_i \in V$ represents task τ_i of which W_i is the worst-case execution cycles (WCEC). We assume that the actual execution cycles of τ_i follows the probability distribution denoted by function $P_i(\cdots)$. Specifically, $P_i(X)$ is the probability that task τ_i executes for X $(1 \leq X \leq W_i)$ cycles. Obviously, $\sum_{x=1}^{W_i} P_i(x) = 1$ and $P_i(W_i) \neq 0$. The corresponding cumulative distribution function is $cdf_i(x) = Prob(X \leq x) = \sum_{j=1}^{x} P_i(j)$ and $cdf_i(0) = 0$. In practice, a histogram is used to represent the probability function considering that a task usually takes millions of cycles. Let the number of bins in the histogram that represents the probability density function be denoted by r_i and denote the bin boundaries by $B_i(k), k = 1, 2, \ldots, r_i$. In this case, function $P_i(\cdot)$ is a function of bin number, that is, $P_i(k)$ $(1 \leq k \leq r_i)$ denotes the probability that task τ_i executes for $B_i(k)$ cycles.

The directed edge e_{ij} represents dependency between task τ_i and τ_j, that is, τ_j is ready to begin execution only after τ_i finishes execution (τ_i is called the predecessor of τ_j and τ_j is called the successor of τ_i). A communication volume v_{ij} is associated with edge e_{ij}, and determines the time and energy cost when τ_i and τ_j are scheduled on two different processors; $v_{ij} = 0$ if the communicating tasks are scheduled on the same processor. That is, we assume that the communication cost is zero if the communicating tasks are scheduled on the same processor. In a task graph, the source is the only vertex that has no predecessors and the sink is the only vertex that has no successors. In streaming applications, the source receives the requests and the sink emits the output of servicing the requests. The period is T (i.e.,

the streaming application is invoked every T time units and thus must sustain a throughput of $\frac{1}{T}$), and the deadline for emitting output (i.e., response time requirement) is D.

There are times when detailed task graph of a streaming application is not available. In this case, the streaming application is simply represented by a single task, which can be regarded as a special case of task graph (singleton task graph) that has only a single vertex.

3.2 SYSTEM MODEL

We consider two system models in this dissertation: uniprocessor and multiprocessor. Each processor in our system models have the ability to dynamically adjust its speed/frequency and voltage. For multiprocessor systems, we consider a typical homogeneous chip multiprocessor (CMP) architecture with distributed memory (Figure 1). Each processor core consists of a processing unit, a local memory, and a switch. We assume that there is an infinite supply of processor cores on the system. This assumption is based the trend that more and more processor cores are available on a single system, as mentioned in Section 2.5.

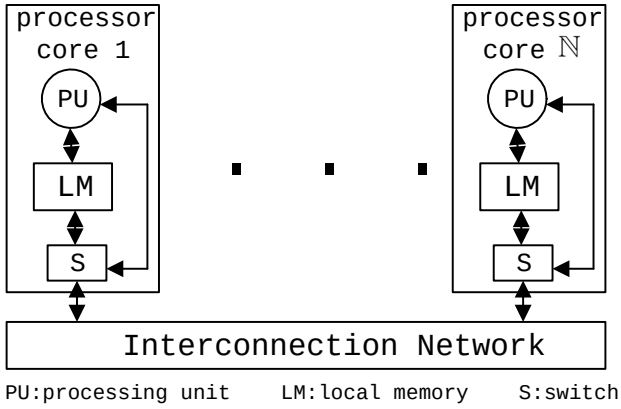


Figure 1: Chip multiprocessor model

We assume that the program for each task in a streaming application is written in stream programming style [58], that is, the program goes through three steps for servicing each request: (i) data gathering from communication network to local memory; (ii) data

processing in local memory; (iii) data (i.e., processing results) dissemination from local memory to communication network.

3.3 PROCESSOR MODEL

We consider two processor models in this dissertation.

3.3.1 Ideal Model

The first model, the *ideal processor model*, assumes that the processor speed can be adjusted continuously from zero to infinity and there is no speed change overhead. The processor power consumption when executing task τ_i at frequency f is

$$p_i(f) = c_0 + c_i f^\alpha$$

where α ($\alpha > 1$) reflects the convex power-frequency relationship, c_0 denotes the processor static power consumption when the processor is idle (i.e., $f = 0$, since the processor is not executing any task and the overhead is ignored), and c_i reflects the effective switching capacitance of task τ_i. This form of analytical power function is due to the fact that dynamic power consumption can be approximately computed by $C_e \times V_{dd}^2 \times f$ (V_{dd} is the supply voltage and C_e is the effective switching capacitance) and the frequency f is almost linearly related to the supply voltage [60]. If a processor (either in uniprocessor or multiprocessor system) is on (or alternatively, active), the amount of the idle power is always consumed in the system. Therefore, the power function of the processor can sometimes be simplified as

$$p_i(f) = c_i f^\alpha$$

in the analysis without affecting the analysis results. We assume that all tasks are CPU-bound. Thus, the execution time of a task is inversely proportional to its operating frequency.

Although the ideal processor model is simplistic, it captures the essence of a DVS system (i.e., the convex power-frequency relationship). Because the simple model is easy to manipulate mathematically, it is possible to derive optimal DVS schemes, which gives us great insight into the problem and provides the basis for designing practical DVS schemes.

3.3.2 Realistic Model

The second model, the *realistic processor model*, considers practical issues. The processor only provides M discrete operating frequencies, $f_1 < f_2 < \cdots < f_M$. All frequencies are efficient, which means that using a frequency to execute a task always results in lower energy consumption than using higher frequencies [46]. The processor power consumption when idle is p_{idle} (the system is not executing any task, and thus consumes constant power (not necessarily $f = 0$ as in the ideal processor model). The processor power consumption when executing task τ_i at frequency f_j is $p_i(f_j)$. As in the case of the ideal processor model, if a processor is active, we sometimes ignore the idle power in deriving DVS schemes and use

$$\hat{p}_i(f_j) = p_i(f_j) - p_{idle}$$

as the power function in the analysis. As in the ideal model, the execution time of a task is inversely proportional to its operating frequency.

In the realistic processor model, when changing the frequency of the processor from f_i to f_j, the time cost is

$$P_T(f_i, f_j) = \xi_1 |f_i - f_j| \tag{3.1}$$

and the energy cost is

$$P_E(f_i, f_j) = \xi_2 |f_i^2 - f_j^2| \tag{3.2}$$

where ξ_1 and ξ_2 are constants determined by the voltage regulators. Equations (3.1) and (3.2) are taken from [15] and are considered to be an accurate modeling of speed change overheads. It is common in the literature (e.g., [47]) to simplify Equations (3.1) and (3.2) by considering the worst-case frequency swing and assuming that $P_T(f_i, f_j) = \xi_1(f_M - f_1)$ and $P_E(f_i, f_j) = \xi_2(f_M^2 - f_1^2)$. Note that $P_T(f, f) = 0$ and $P_E(f, f) = 0$, where $f \in \{f_1, f_2, \ldots, f_M\}$.

3.4 COMMUNICATION MODEL

For multiprocessor systems, we adopt a linear communication cost model, that is, when transferring B bits of data, the communication delay is $t_p + \lambda B$ and the communication energy is γB, where t_p is the propagation delay, λ is the reciprocal of the operating data rate of the interconnection network, and γ is the energy spent to transfer one bit of data. In this dissertation, we assume fixed data rate, that is, fixed γ and λ. Although the adopted linear model does not account for network contention, it was shown to work very well for high-bandwidth interconnection networks [49], which is typical under current multi-core processor technology (e.g., CELL [1]).

3.5 PROBLEM DESCRIPTION

The problems that this dissertation addresses are essentially constrained optimization problems. All problems share the same goal, which is to minimize the energy consumption of a targeted streaming application. They also share the same constraints, namely two QoS requirements of throughput and response time. The requests are coming at a rate of one request every T seconds, and thus requiring that the streaming application can sustain a throughput of $\frac{1}{T}$. Each request is required to be processed in a time not greater than D, and thus requiring that the streaming application responds to a request within time D. The throughput performance is measured only when the response time limit is met, and the response time is measured only when the throughput requirement is satisfied.

However, the problems differ in their conditions, which can be categorized through three dimensions. The first dimension is the underlying computing platform : uniprocessor and multiprocessor. The second dimension is the characterization of the task workload : deterministic and stochastic. In the deterministic case, each task consumes its worst-case execution cycles (WCEC) while in the stochastic case, the number of execution cycles of each task is variable and the variability can be captured by a probability distribution function. The third dimension is the type of task graphs : general task graphs and singleton task

graphs (only one vertex). Eight problems can be identified through these three dimensions. Table 1 shows these eight problems and where they are solved. The name of each problem consists of four parts separated by dash, with the first part being STREAM and the next three parts corresponding to the aforementioned three dimensions. Next, we describe these problems in detail.

Table 1: The eight optimization problems under consideration

STREAM-	Deterministic		Stochastic	
	Single Tasks	Task Graphs	Single Tasks	Task Graphs
Uniprocessor	UP-D-ST ([7])	UP-D-TG ([7])	UP-S-ST (Section 4.3.1)	UP-S-TG (Section 4.3.2&4.3.3&4.4)
Multiprocessor	MP-D-ST (Section 5.2)	MP-D-TG (Section 5.3)	MP-S-ST (Section 5.2)	MP-S-TG (Section 5.3)

3.5.1 Uniprocessor Scheduling Problems

In uniprocessor systems, there is no cost associated with communication among tasks. The main focus is to save processor dynamic energy consumption through DVS. This is why the scheduling algorithms for uniprocessor systems are also called *DVS schemes*. Furthermore, for uniprocessor systems, the two QoS requirements essentially collapse into one in the sense that we only need to deal with one requirement. This is because if the required response time D is less than the request interarrival time T, each request must be processed in time D; if $D > T$, for a total of $\mathbb{N}$ requests, we have a total of $\mathbb{N} \cdot T + D \approx \mathbb{N} \cdot T$ ($\mathbb{N}$ is a large number) time to process all these requests, which means each request must be processed in time T. Thus, the deadline for processing a request is $min(D, T)$. Without loss of generality, we assume that $D \leq T$ so that we can just consider the response time requirement.

3.5.1.1 The STREAM-UP-D-ST and STREAM-UP-D-TG Problems Both the STREAM-UP-D-ST and STREAM-UP-D-TG problems deal with scheduling a streaming

application whose workload is deterministic on a uniprocessor system. Their difference is whether or not the streaming application has a detailed task graph. However, because the workload is deterministic and there is no cost associated with inter-task in-processor communication, the STREAM-UP-D-TG problem can be reduced to the STREAM-UP-D-ST problem by transforming the task graph into a single task. The STREAM-UP-D-ST problem has been well studied. If the workload of a streaming application is deterministic, one can always use a constant processor speed to execute that streaming application to achieve the minimum energy consumption without missing the deadline. This is due to the convexity of the processor power function. In fact, this is a well-establish result in energy-aware scheduling theory. Interested readers can refer to [7] for a formal proof. Thus, the solution to the problems of STREAM-UP-D-ST and STREAM-UP-D-TG will not be considered in this dissertation.

3.5.1.2 The STREAM-UP-S-ST Problem The STREAM-UP-S-ST problem deals with scheduling a streaming application represented by a single task whose workload is stochastic on a uniprocessor system. More often than not, the computational requirement of a task is variable and not known a-priori. One could attempt to predict the computational requirement of the task. However, for many streaming applications (e.g., MPEG decoder), the computational requirement cannot be predicted based on the recent history. The variability and unpredictability of the workloads are mainly caused by different inputs to tasks, and possibly by randomization inside tasks. For variable workloads, we focus on *stochastic DVS schemes* that use probability distribution functions to capture the variability of the workloads. Thus, the goal of such DVS schemes is to minimize the *expected* energy consumption. The basic question of the STREAM-UP-S-ST problem is: given a task and a deadline, how to decide the execution speed for the task such that the expected energy consumption is minimized while meeting the deadline? Note that the solution to the STREAM-UP-S-ST problem can be applied to frame-based real-time systems with only a single task. This is because the processing of a request can be viewed as a task and the response time can be treated as the frame length. In Chapter 5, the solution to the STREAM-UP-S-ST problem also serves as an important building block for the solutions to scheduling problems in

multiprocessor systems.

3.5.1.3 The STREAM-UP-S-TG Problem The STREAM-UP-S-TG problem deals with scheduling a streaming application represented by a task graph whose workload is stochastic on a uniprocessor system. Topological sort can be performed on the task graph to obtain a chain of tasks to be executed in the system in succession. The basic question of this problem is: given a chain of tasks and a deadline, how to decide the execution speed for each task such that the expected energy consumption of all tasks is minimized while all tasks will finish by the deadline? Since the workload of each task is variable, dynamic slack reclamation plays an important role in this problem. By knowing more information (i.e., the probability distribution of the computational requirement of each task), more energy savings are expected to be obtained than only knowing the probabilistic information of the whole task graph. The solution to this problem can be applied to general frame-based real-time systems.

3.5.2 Multiprocessor Scheduling Problems

Multiprocessor scheduling problems differ from uniprocessor scheduling problems mainly in two aspects. First, unlike uniprocessor systems, we have the freedom of choosing the number of processors to execute a streaming application in multiprocessor systems. The two power management mechanisms (i.e., on-off and DVS) must be used in the scheduling because of the trade-off between static and dynamic power consumption. Thus, determining the appropriate number of processors is one of the keys in solving multiprocessor scheduling problems. Second, because parallel processing and pipelining techniques can be used in multiprocessor systems, we can take advantage of the case where response time requirement D is greater than request interarrival time T to save more energy.

3.5.2.1 The STREAM-MP-D-ST Problem The STREAM-MP-D-ST problem deals with scheduling a streaming application represented by a single task whose workload is deterministic on a multiprocessor system. Since the streaming application is only represented

by a single task, the whole application must be executed in a single processor. However, we could potentially execute different instances of the streaming application on different processors to serve different requests. Specifically, we have a processor that acts as a master to receive requests and distribute them in a round-robin fashion to other processors, each acting as a slave and running an instance of the streaming application. The master can be placed on the administrative processor (e.g., the PPE in CELL [1]), or with a slave on a processor because the master has very light workload. Therefore, we ignore the energy consumption of the master. Thus, the basic question of the STEAM-MP-D-ST problem is how to decide the number of processors (slaves) and the speed for each processor.

3.5.2.2 The STREAM-MP-D-TG Problem The STREAM-MP-D-TG problem deals with scheduling a streaming application represented by a task graph whose workload is deterministic on a multiprocessor system. Since the streaming application has a detailed task graph exposing the parallelism inside the application, classic pipelining and parallel processing techniques can be applied in multiprocessor task graph scheduling. Pipelining is used to exploit the parallelism in time (indicated by predecessor and successor relationship in task graphs) and parallel processing is employed to take advantage of the parallelism in space (indicated by sibling relationship in task graphs). The basic questions of the STREAM-MP-D-TG problem are how to decide: (i) the number of active processors to execute the task graph; (ii) the mapping from tasks to active processors; (iii) the execution speed of each task. Note that these three questions are correlated and will be addressed simultaneously.

3.5.2.3 The STREAM-MP-S-ST and STREAM-MP-S-TG Problems These two problems are similar to their deterministic counterparts except that the computational requirement of each task is variable and unpredictable. As in the uniprocessor case, the variability of the workload can be captured by a probability distribution function, and the objective is to minimize the expected energy consumption. Dynamic slack reclamation technique is indispensable in dealing with a stochastic workload. Thus, the scheduling algorithms for these two problems must consider reclaiming dynamic slack generated in the system. The basic question of these two problems is whether we can extend the algorithms for their de-

Table 2: Synthetic task graphs

Task graph	# of tasks	# of edges
kseries_parallel	20 - 62	19 - 61
creds1	9 - 13	10 - 18
simple	11 - 24	12 - 32
kbasic_tables	19	21
kseries_parallel_xover	21 - 38	24 - 41
bugtest	49	60
kbasic_task	18 - 64	18 - 79
kextended	21 - 23	25 - 28
packets	6 - 8	5 - 8

terministic counterparts or we need to design the algorithms for these two problems from scratch.

3.6 EVALUATION METHODOLOGY

The solutions to the problems described in Section 3.5 are evaluated through simulations. The purpose of the evaluation is two-fold. First, simulation results can provide insight into the solutions. Second, we want to quantify the gains of our solutions over previously known solutions.

Next, we describe the workload generation and process models used in our simulations.

3.6.1 Workload Generation

Both synthetic and real-world workloads are used in the simulations. Synthetic task graphs (Table 2) are generated by TGFF v3.0 [53] using the sample input files that come with the

software package. For stochastic workloads, we use six representative probability distributions (Figure 2) to generate the number of execution cycles of a task. The distributions include one uniform distribution, three unimodal distributions, and two bimodal distributions.

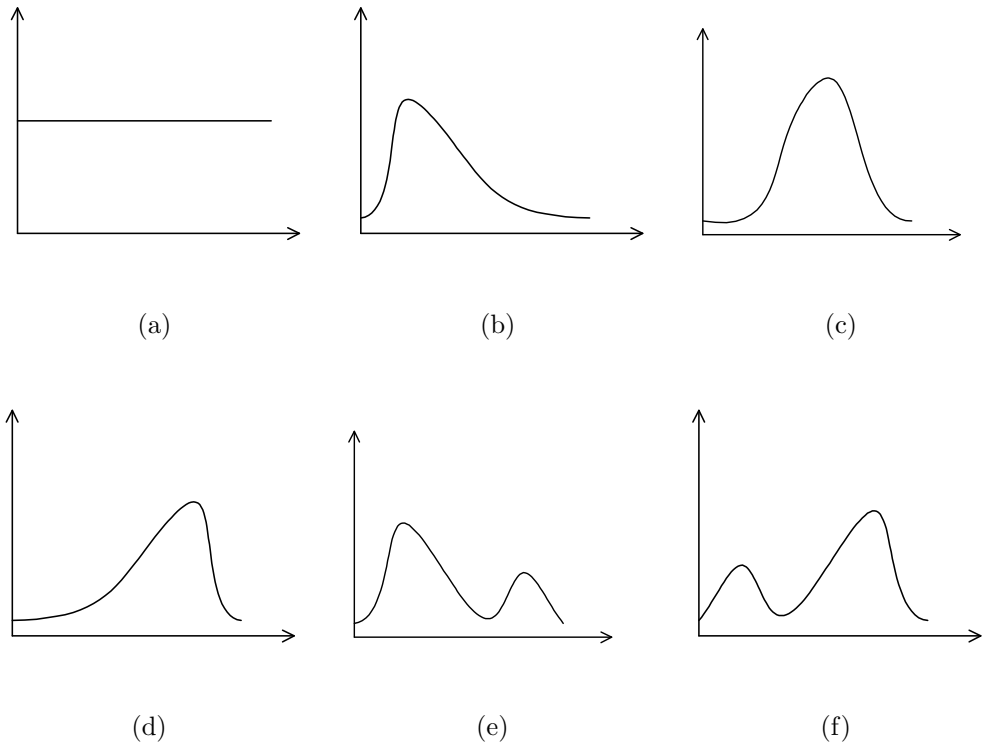


Figure 2: Probability functions: uniform, unimodal1, unimodal2, unimodal3, bimodal1, bimodal2 (from left to right). The Y-axis is probability and the X-axis is number of execution cycles.

For real-world task graphs, we used automatic target recognition (ATR) [43], which is a streaming application that does pattern matching of targets in images/pictures. In ATR, the regions of interest (ROI) in one image are detected and each ROI is compared with a number of stored templates. The number of targets detected in each frame varies from 0 to 8. Image processing time is proportional to the number of detections within an image. A typical platform for ATR is unmanned autonomous vehicle (UAV). ATR must sustain a required incoming rate of images and process each image within a required amount of time

to meet UAV mission requirement[1].

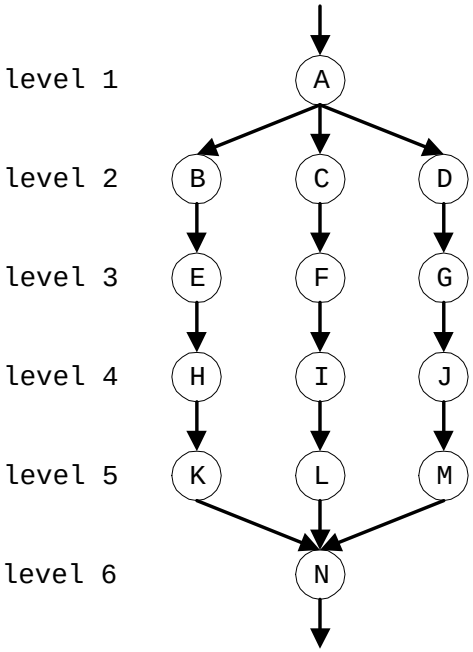


Figure 3: Task graph of ATR when three targets are detected

The task graph of ATR is different for different number of target detections in an image. Figure 3 shows the one corresponding to 3 target detections. Each of the three paths, $B \rightarrow E \rightarrow H \rightarrow K$, $C \rightarrow F \rightarrow I \rightarrow L$, and $D \rightarrow G \rightarrow J \rightarrow M$, corresponds to the processing for one target detection. Note that these three paths have the same structure and workload since ATR does the same processing for different target detections. .

3.6.2 Processor Models

We use the following three processor models (all belong to the realistic model) in our simulations.

1. Synthetic processor. It strictly conforms to the $p(f) = f^3$ power-frequency relation and has 10 discrete frequencies ranging from 100MHz to 1GHz with 100MHz step; its idle power is zero and there is no speed change overhead.

[1]The throughput and response time QoS requirements are explicitly specified in the ATR documentation

2. Intel XScale (Table 3) [64]. The idle power of XScale is 40 mW [19], which is one half the power consumed at the minimum frequency. Figure 4(a) shows that the approximate analytical power function obtained by curve fitting is a good approximation of the actual power function. The time and energy penalties for each speed change are reported as 12μs and 1.2μJ, respectively, in [63]. We assume that these numbers are worst-case speed change overheads, which are equivalent to the overheads incurred when changing from the minimum speed to the maximum speed in our power model described in Section 3. Accordingly, we derived the values of the constants ξ_1 and ξ_2 in Equation (3.1) - (3.2) to be used in our experiments.

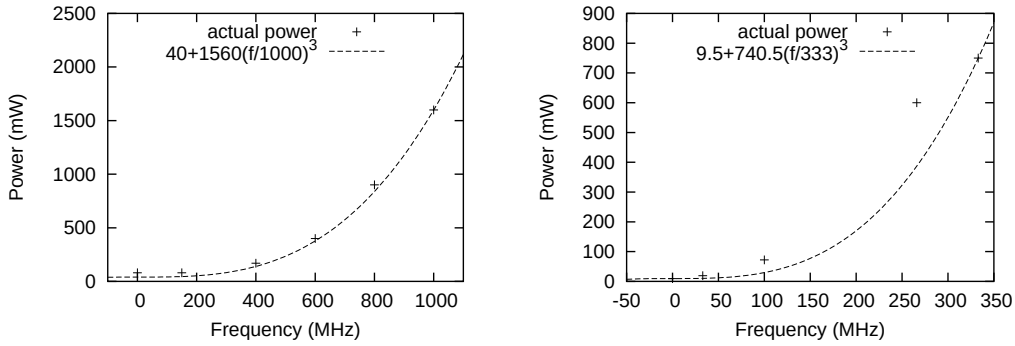
3. IBM PowerPC 405LP (Table 4) [54]. The approximate analytical power function obtained by curve fitting is shown in Figure 4(b), which indicates that the approximation is not as good as XScale. The idle power is assumed to be half the power consumed at the minimum frequency. The time and energy penalties for each speed change are reported as 1ms and 750μJ, respectively, in [54]. We also translated these numbers into our power model as in the case of XScale.

Table 3: XScale speed settings and power consumptions

Speed (MHz)	Idle	150	400	600	800	1000
Voltage (V)	0.75	0.75	1.0	1.3	1.6	1.8
Power (mW)	40	80	170	400	900	1600

Table 4: PowerPC 405LP speed settings and power consumptions

Speed (MHz)	Idle	33	100	266	333
Voltage (V)	1.0	1.0	1.0	1.8	1.9
Power (mW)	9.5	19	72	600	750



(a) XScale

(b) PowerPC 405LP

Figure 4: Approximate analytical power function vs. actual power function

The power consumptions listed in Tables 3 and 4 are obtained by measuring the processor power when running certain benchmarks. In practice, different tasks have different instruction mixes, thus resulting in different dynamic power consumptions. We associate each task with a *power scaling factor* to simulate this reality. For example, if a task's power scaling factor is 0.9, it will consume $40 + (400 - 40) \times 0.9 = 364$ mW when executing at frequency 600MHz for XScale.

4.0 SCHEDULING IN UNIPROCESSOR SYSTEMS

4.1 OVERVIEW

In this chapter, we consider energy-aware uniprocessor scheduling problems for streaming applications. Since the STREAM-UP-D-ST and STREAM-UP-D-TG problems have been well studied, we will focus on the problems of STREAM-UP-S-ST and STREAM-UP-S-TG (i.e., scheduling stochastic workload in uniprocessor systems).

The problems of STREAM-UP-S-ST and STREAM-UP-S-TG are closely related to *frame-based hard* real-time systems, which are special cases of periodic real-time systems. In frame-based real-time systems, all tasks share the same period (also called the *frame*) and deadlines are equal to the end of the period. In each frame, tasks are executed in a predetermined fixed order. Thus, the STREAM-UP-S-ST problem can be modeled as a frame-based system in which there is only a single task, while the STREAM-UP-S-TG can be modeled as a regular frame-based system in which the task execution order is obtained by using topological sort on the task graph. For frame-based systems, the problem of designing a DVS scheme can be reduced to determining the amount of time allotted to a task (and accordingly, deciding the speed(s) to execute it) before it is dispatched to the system.

In the STREAM-UP-S-ST and STREAM-UP-S-TG problems, the workload of streaming applications is variable and unpredictable. In these applications, tasks usually run for less than their worst-case execution cycles (WCEC[1]), creating the opportunity for dynamic slack reclamation to slow down future tasks. Furthermore, the actual number of execution cycles of tasks are unknown and cannot be predicted before execution. Because of the above

[1]The traditional worst-case execution time (WCET) in real-time systems is defined as the running time at the maximum processor speed when a task takes its WCEC.

characteristics, it is impossible for any DVS scheme to guarantee to minimize the energy consumption in the system. However, if the variability of the workloads can be captured by the probability distribution of the computational requirement of each task in the system (e.g., through profiling), it is possible to design DVS schemes that minimize the expected system energy consumption. This means that for large number of frames, such DVS schemes will achieve the most total energy savings even though they do not necessarily result in lower energy consumption than other DVS schemes for any given frame.

In this chapter, we investigate DVS schemes for frame-based hard real-time systems with the goal of minimizing the expected energy consumption in the system while meeting the deadlines, given probabilistic information of the workloads. We carry out the investigation in two dimensions. The first dimension is the DVS strategy, which can be categorized as *inter-task*, *intra-task*, and *hybrid* DVS (refer to Section 2.4 for definitions of these strategies). Intra-task DVS schemes are solutions to the STREAM-UP-S-ST problem, while inter-task and hybrid DVS schemes are solutions to the STREAM-UP-S-TG problem. The second dimension is the processor power model, for which we have the ideal and realistic models. Because of the great simplicity of the ideal model, it is possible to derive elegant and optimal DVS schemes through which we gain insight into the problems that we are dealing with. Furthermore, the DVS schemes derived under the ideal processor model could serve as the basis for deriving DVS schemes under the *realistic* model. *Our ultimate goal is to obtain good DVS schemes under the realistic model.*

Table 5 shows the road map of our investigation in this chapter. We first present optimal DVS schemes for each DVS strategy under the ideal processor model in Section 4.2.1, 4.2.2, and 4.2.3. We then give a unified view of all these optimal schemes in Section 4.2.4. DVS schemes for each DVS strategy under the realistic processor model are presented in Section 4.3. These schemes are designed using different approaches. In Section 4.4, we propose a unified approach for obtaining optimal (or provably close to optimal) DVS schemes for all DVS strategies under the realistic processor model.

Table 5: The road map of our investigation; cited work was done by other researchers prior to this dissertation

Problem	STREAM-UP-S-ST	STREAM-UP-S-TG	
Model/DVS strategy	Intra-task DVS	Inter-task DVS	Hybrid DVS
Ideal model	PACE [40] (Section 4.2.1)	OITDVS (Section 4.2.2)	GOPDVS [72] (Section 4.2.3)
Realistic model	PPACE (Section 4.3.1)	PITDVS PITDVS2 (Section 4.3.2)	PGOPDVS PIT-PPACE (Section 4.3.3)
	A unified approach (Section 4.4)		

4.2 SOLUTIONS UNDER IDEAL PROCESSOR MODEL

4.2.1 Optimal Intra-Task Scheme

The ideal processor model assumes the ability to set speed/frequency for each cycle executed by a task, since there is no speed change overhead. Thus, the problem of deriving an intra-task DVS scheme is to find the execution speed for each cycle of a task (also called the *speed schedule* of the task) to minimize the expected energy consumption while ensuring that the task will finish by the deadline D. An intra-task DVS scheme is of great importance because it could be used as a building block for general frame-based real-time systems containing multiple tasks: a time D_i is allotted to each task, and the intra-task DVS scheme is applied for each task. It can also be used as the DVS scheme for a special case of frame-based systems, where there is only a single task in a frame (e.g., for specialized embedded devices like MP3 players).

The optimal intra-task DVS scheme, PACE, was derived in [40]. We will briefly review its derivation for completion. Since there is only a single task, we will omit the subscripts

of the parameters of the task unless confusion arises. Let the execution speed of the i^{th} cycle of task τ be denoted by s_i. The problem can be formally expressed by the following mathematical program [40]

$$\texttt{Minimize} \quad \sum_{1 \le i \le W} F_i \cdot c \cdot s_i^{\alpha-1} \tag{4.1}$$

$$\texttt{Subject to} \quad \sum_{1 \le i \le W} \frac{1}{s_i} \le D \tag{4.2}$$

where $F_i = 1 - cdf(i-1)$ represents the probability of executing the i^{th} cycle of task τ. Note that c in (4.1) reflects the effective switching capacitance of task τ. By using Lagrangian technique [36] or Jensen's inequality [35], the optimal solution to the problem in (4.1)-(4.2) is

$$s_i = \frac{\sum_{j=1}^{W} F_j^{\frac{1}{\alpha}}}{F_i^{\frac{1}{\alpha}} D} \tag{4.3}$$

and the optimal expected energy consumption is

$$e^* = \frac{c \sum_{j=1}^{W} F_j^{\frac{1}{\alpha}}}{D^{\alpha-1}} \tag{4.4}$$

Several claims can be made based on (4.3) and (4.4): (i) the optimal expected energy consumption of a task is inversely proportional to $D^{\alpha-1}$; (ii) $s_1 \le s_2 \le \ldots \le s_W$ [40] since $cdf(\cdot)$ is a nondecreasing function. Using lower speeds for early cycles makes perfect sense because early cycles have higher probability to be executed than later cycles. (iii) when changing speed during the execution of a task, the optimal speed to be used is always inversely proportional to the remaining time until the deadline. This claim is not straightforward and needs a little clarification. Let $\eta_i = F_i^{\frac{1}{\alpha}} / \sum_{j=1}^{W} F_j^{\frac{1}{\alpha}}$ and thus (4.3) can be rewritten as $s_i = \frac{1}{\eta_i D}$. One can view η_i as the fraction of the time D to be allotted to the i^{th} cycle. Claim (iii) is obviously true for executing the 1^{st} cycle and we will show that it is also true for the i^{th} cycle, where $i > 1$. Before executing the i^{th} cycle, the remaining time until the deadline is $D - \sum_{j=1}^{i-1} \eta_j D = D(1 - \sum_{j=1}^{i-1} \eta_j)$ and the time allotted to the i^{th} cycle is $\eta_i D$. If β_i is the fraction of the remaining time dedicated to τ_i, that is,

$$\beta_i = \frac{\eta_i}{1 - \sum_{j=1}^{i-1} \eta_j}$$

we have

$$s_i = \frac{1}{\beta_i D(1 - \sum_{j=1}^{i-1} \eta_j)}$$

Thus claim (iii) above can be made for all cycles of the task.

4.2.2 Optimal Inter-Task Scheme

The key for inter-task DVS schemes is how to allot the slack in the system (or equivalently, the available system time) to the tasks. In doing so, one needs to take into consideration the variability of the workload and possible dynamic slack reclamation. Next, we will describe the derivation of our optimal scheme under the ideal processor model.

The optimal DVS scheme is based on an important property that the optimal expected energy consumption of a sequence of tasks is inversely proportional to $t^{\alpha-1}$, where t is the amount of allotted time. To introduce this property, we start by establishing a preliminary lemma on the energy consumption of a single task.

Lemma 1. *If a task τ cannot change speed during the execution, its optimal expected energy consumption is inversely proportional to $t^{\alpha-1}$, where t is the amount of time allotted to execute τ.*

Proof. Suppose that W is the worst-case number of execution cycles of τ, and $P(x)$ is the probability that τ executes for x cycles. Obviously, we should use the lowest possible speed, $\frac{W}{t}$, such that τ will finish in time t in the worst case. Therefore, the optimal expected energy consumption of executing τ is

$$\sum_{x=1}^{W} P(x) p\left(\frac{W}{t}\right) \frac{x}{\frac{W}{t}} = \frac{W^{\alpha-1} c \sum_{x=1}^{W} P(x) x}{t^{\alpha-1}} \tag{4.5}$$

which is inversely proportional to $t^{\alpha-1}$ (recall that $p(\cdot)$ is the power function). □

Interestingly, the result of Lemma 1 still holds for multiple tasks.

Theorem 1. *If tasks cannot change speed during their execution, the optimal expected energy consumption of executing N tasks $\tau_1, \tau_2, \ldots, \tau_N$ consecutively is inversely proportional to $t^{\alpha-1}$, where t is the amount of time allotted to execute these tasks.*

Proof. Suppose that the worst-case number of execution cycles of τ_i is W_i, and the probability that τ_i executes for x cycles is $P_i(x)$.

Let the *optimal* expected energy consumption of executing tasks $\tau_i, \tau_{i+1}, \ldots, \tau_N$ consecutively with allotted time d be denoted by $E(i,d)$. We will prove by induction that $E(1,d)$ is inversely proportional to $d^{\alpha-1}$.

The induction is on i. The base case for $E(N,d)$ is obviously true by Lemma 1.

In the induction step, assume that $E(i+1,d)$ is inversely proportional to $d^{\alpha-1}$, that is, $E(i+1,d) = \frac{C_{i+1}}{d^{\alpha-1}}$, where C_{i+1} only depends on $W_{i+1}, \ldots, W_N$ and $P_{i+1}(x), \ldots, P_N(x)$. To compute $E(i,d)$, we first compute a helper function, $\tilde{E}(i,d,\beta)$, which denotes the expected energy consumption of executing tasks $\tau_i, \tau_{i+1}, \ldots, \tau_N$ with allotted time d when allotting a fraction β $(0 < \beta \leq 1)$ of time d to task τ_i and allotting the remaining time, $(1-\beta)d$, to tasks $\tau_{i+1}, \ldots, \tau_N$. The running speed for τ_i is obviously $\frac{W_i}{\beta d}$, and the time left for executing tasks $\tau_{i+1}, \tau_{i+2}, \ldots, \tau_N$ is $d - x/\frac{W_i}{\beta d}$ when τ_i executes x cycles. Therefore,

$$\begin{aligned}
\tilde{E}(i,d,\beta) &= \sum_{x=1}^{W_i} P_i(x) \left(p\left(\frac{W_i}{\beta d}\right) x/\frac{W_i}{\beta d} + \frac{C_{i+1}}{(d - x/\frac{W_i}{\beta d})^{\alpha-1}} \right) \\
&= \sum_{x=1}^{W_i} P_i(x) \left(x c_i \left(\frac{W_i}{\beta d}\right)^{\alpha-1} + \frac{C_{i+1}}{(d - \frac{x\beta d}{W_i})^{\alpha-1}} \right) \\
&= \frac{\sum_{x=1}^{W_i} P_i(x) \left(x c_i \left(\frac{W_i}{\beta}\right)^{\alpha-1} + \frac{C_{i+1}}{(1-\frac{x\beta}{W_i})^{\alpha-1}} \right)}{d^{\alpha-1}} \\
&= \frac{\frac{W_i^{\alpha-1} c_i \sum_{x=1}^{W_i} P_i(x) x}{\beta^{\alpha-1}} + C_{i+1} \sum_{x=1}^{W_i} \frac{P_i(x)}{(1-\frac{x\beta}{W_i})^{\alpha-1}}}{d^{\alpha-1}} \\
&= \frac{\phi_i(\beta) + \varphi_i(\beta)}{d^{\alpha-1}}
\end{aligned}$$

where $\phi_i(\beta) = \frac{W_i^{\alpha-1} c_i \sum_{x=1}^{W_i} P_i(x) x}{\beta^{\alpha-1}}$ and $\varphi_i(\beta) = C_{i+1} \sum_{x=1}^{W_i} \frac{P_i(x)}{(1-\frac{x\beta}{W_i})^{\alpha-1}}$.

Let

$$C_i = \min_{0<\beta\leq 1} \left(\phi_i(\beta) + \varphi_i(\beta)\right)$$

and

$$\beta_i = \operatorname*{argmin}_{0<\beta\leq 1} \left(\phi_i(\beta) + \varphi_i(\beta)\right)$$

Then, the minimum expected energy consumption, $E(i, d)$, is

$$E(i, d) = \tilde{E}(i, d, \beta_i) = \frac{C_i}{d^{\alpha-1}}$$

Substituting 1 for i and t for d in $E(i, d)$ will complete the proof. □

Theorem 1 shows that the optimal expected energy consumption of a sequence of tasks is of the same form as that of a single task, that is, both are inversely proportional to the $(\alpha - 1)$st power of the allotted time. This is a very powerful result because it can be used to optimize the expected energy consumption for a sequence of tasks. When a sequence of tasks is to be executed, the tasks are partitioned into two parts: the first task and the rest of the tasks, which can be treated as one supertask. Thus, the problem of allotting time to multiple tasks is reduced to allotting time to just two tasks, which can be efficiently solved thanks to the nice form of the power function in the ideal processor model. In fact, this is the basic idea of the proof of Theorem 1.

The proof of Theorem 1 indicates that in order to minimize the expected energy consumption of executing a sequence of tasks within a given amount of time t, one should allot to the first task a *fixed fraction* of time t and set the speed such that the first task is guaranteed to finish within the time allotted to it in the worst case. When the first task finishes, the same procedure can be applied recursively to the rest of the tasks, with the time left until the deadline.

The proof of Theorem 1 also leads to the computation of the time allocation fraction for each task. As in the proof, let C_i denote the constant in the optimal expected energy consumption of executing $\tau_i, \tau_{i+1}, \ldots, \tau_N$ consecutively and β_i denote the optimal time allocation fraction for τ_i. We compute C_i and β_i in the reverse order. That is, first compute C_N, β_N, then C_{N-1}, β_{N-1}, ... , and last C_1, β_1. The efficiency of the algorithm depends on how to find the minimum value of $\phi_i(\beta) + \varphi_i(\beta)$. We do not have a closed form formula for it. However, by computing the first and second derivatives of $\phi_i(\beta)$ and $\varphi_i(\beta)$, we find that $\phi_i(\beta)$ is a convex decreasing function and $\varphi_i(\beta)$ is a convex increasing function. It is easy to show that $\phi_i(\beta) + \varphi_i(\beta)$ is a convex function with only one global minimum when $0 < \beta \leq 1$. Thus, finding the minimum value of $\phi_i(\beta) + \varphi_i(\beta)$ can be efficiently solved to

any desirable precision by using many existing numerical optimization methods, such as the gradient descent. The algorithm for computing the time allocation fractions, $\beta_1, \beta_2, \ldots, \beta_N$, is shown in Algorithm 4.1.

Algorithm 4.1 OITDVS-Offline$([W_1, W_2, \ldots, W_N], [P_1(x), P_2(x), \ldots, P_N(x)]$

1: $\beta_N := 1$
2: $C_N := W_N^{\alpha-1} c_N \sum_{x=1}^{W_N} P_N(x)x$
3: **for** $i := N-1$ downto 1 **do**
4: $\quad F(\beta) = \left(\frac{W_i}{\beta}\right)^{\alpha-1} c_i \sum_{x=1}^{W_i} P_i(x)x + \sum_{x=1}^{W_i} \frac{P_i(x)C_{i+1}}{(1-\frac{x\beta}{W_i})^{\alpha-1}}$
5: $\quad C_i := \min_{0<\beta\leq 1} F(\beta)$
6: $\quad \beta_i := \operatorname{argmin}_{0<\beta\leq 1} F(\beta)$
7: **end for**
8: **return** $[\beta_1, \beta_2, \ldots, \beta_N]$

Assuming that the probability distributions are stable, the computation of β_i can be done offline for all the values of i. The online part, to select the speed at which a task needs to execute, is straightforward: when starting the execution of task τ_i and having time d left for executing $\tau_i, \tau_{i+1}, \ldots, \tau_N$, we allocate time $\beta_i d$ to τ_i and the speed is set to $\frac{W_i}{\beta_i d}$.

The offline and online parts described above form the **O**ptimal **I**nter-**T**ask **DVS** scheme under the ideal processor model, which we call the OITDVS scheme.

4.2.3 Optimal Hybrid Scheme

Since intra-task DVS only focuses on speed scheduling within a single task, one would wonder whether it could be applied to the case of multiple tasks. A naive extension to an intra-task DVS scheme (e.g., PACE [40]) would treat all the tasks as a single supertask and derive its parameters (WCEC and probability function) from those of the original tasks. Unfortunately, for this supertask, using PACE will generally result in more energy consumption than the DVS schemes that simply use dynamic slack reclamation (e.g., the Proportional scheme [48]). Appendix B provides an illustrative example to clarify this point. The reason why this naive extension fails is that treating all tasks as a single supertask results in loss of information (e.g., the naive extension cannot determine when tasks terminate), losing the opportunity for

dynamic slack reclamation, which is an indispensable element for DVS schemes for multiple tasks. Therefore, intra-task DVS needs to be combined with inter-task DVS for possible further energy savings over DVS schemes that use only intra-task DVS or inter-task DVS alone.

The optimal hybrid DVS scheme was proposed by Zhang et al. in [72] and it is called Global OP-DVS scheme (OP stands for optimal). Throughout this dissertation, we call this scheme GOPDVS. The derivation of GOPDVS is similar to that of OITDVS described in Section 4.2, as a task can be treated as W one-cycle subtasks, where W is the WCEC of the task. For the sake of completeness of this dissertation, we rewrite the offline part of GOPDVS using our notation in Algorithm 4.2. In contrast to our OITDVS scheme, the output of the procedure GOPDVS-offline is the time allocation fractions β_{ij} for the j^{th} cycle of task τ_i, where $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, W_i$. More technical details can be found in [72].

The online part of GOPDVS goes as follows: when starting the execution of the j^{th} cycle of task τ_i and having time d remaining in the frame, allocate time $\beta_{ij}d$ to the j^{th} cycle of task τ_i and the speed is set to $\frac{1}{\beta_{ij}d}$.

Algorithm 4.2 GOPDVS-Offline$([W_1, W_2, \ldots, W_N], [P_1(x), P_2(x), \ldots, P_N(x)])$

1: $C_{N+1} := 0$
2: **for** $i := N$ downto 1 **do**
3: $\quad C := C_{i+1}$
4: $\quad$ **for** $j := W_i$ downto 1 **do**
5: $\quad\quad \beta_{ij} := \frac{\sqrt[\alpha]{c_i}}{\sqrt[\alpha]{c_i} + \sqrt[\alpha]{C}}$
6: $\quad\quad q := \frac{1 - cdf_i(j-1)}{1 - cdf_i(j-2)}$ {let $cdf_i(-1) = 0$}
7: $\quad\quad C := q(\sqrt[\alpha]{c_i} + \sqrt[\alpha]{C})^{\alpha} + (1 - q)C_{i+1}$
8: $\quad$ **end for**
9: $\quad C_i := C$
10: **end for**
11: **return** $[\beta_{ij}]$, $i = 1, 2, \ldots, N, j = 1, 2, \ldots, W_i$

4.2.4 A Unified View

We have presented the optimal DVS schemes for different DVS strategies under the ideal processor model. All of them are targeted at frame-based real-time systems. PACE works for frame-based systems with only a single task, and OITDVS and GOPDVS work for general frame-based systems. In fact, PACE is a special case of GOPDVS.

For frame-based real-time systems, the optimal schemes using different DVS strategies look surprisingly similar, which can be attributed to the assumption of unrestricted continuous frequency, zero speed change overhead, and the nice form of the power-frequency relation. We give a unified view of these optimal DVS schemes in order to appreciate their commonality.

1. The optimal expected energy consumption of a frame, whether using a fixed speed or intra-task DVS for each individual task, is inversely proportional to $D^{\alpha-1}$, where D is the frame length;

2. When scheduling speed changes in a frame, whether at the task boundary in inter-task DVS or for each cycle in intra-task DVS, the optimal speed to be used is inversely proportional to the remaining time in the frame;

3. If a task is executed using intra-task DVS, no matter what position it is in the frame, the optimal speed is always nondecreasing during the execution of the task to minimize the expected energy consumption.

It is noteworthy that although the optimal hybrid DVS scheme achieves better energy savings than the optimal inter-task DVS scheme for general frame-based systems, inter-task DVS schemes are easier to implement than hybrid DVS schemes. In hybrid DVS schemes, timer-like interrupt mechanisms are needed to perform speed changes during the execution of a task.

4.3 SOLUTIONS UNDER REALISTIC PROCESSOR MODEL

In the previous section, we have seen that elegant and optimal DVS schemes exist for different DVS strategies under the ideal processor model. However, the assumptions of the ideal processor model are oversimplified and do not hold in practice. This implies that those optimal DVS schemes might be problematic when used in the real world. In this section, we investigate DVS schemes for different DVS strategies under the realistic processor model.

There are generally two approaches in designing DVS schemes under the realistic processor model. The first approach is to patch the DVS schemes obtained under the ideal processor model, while the second approach is to design DVS schemes considering, from the onset, the realistic model. The advantage of the first approach is its simplicity but it could be far from optimal in terms of energy savings due to its post-processing nature. The second approach is expected to outperform the first approach but is also expected to have higher complexity because the realistic processor model is more complicated than the ideal processor model. Both approaches will be discussed below.

4.3.1 The Intra-task Scheme

To derive DVS schemes under the realistic processor model, simply patching the speed schedule obtained from the ideal processor model may result in a significant deviation from the optimal solution. Thus, our solution, PPACE (Practical PACE), is to attack the problem under the realistic processor model directly.

4.3.1.1 Problem Formulation For the ideal processor model, the speed for *each cycle* of a task is computed, which is obviously too overwhelming considering that a task usually takes millions of cycles. Furthermore, the ability to change speed for any cycle is unreasonable because of the speed change overhead. Real-world operating systems have a granularity requirement for changing speeds [3, 41]. Thus, we need a schedule with a limited number of *speed scaling points*at which speed may change. This implies that the speed remains constant between any two adjacent speed scaling points. As in Section 4.2.1, we will omit

the subscripts of the parameters of the task unless confusion arises. We denote the i^{th} speed scaling point of task τ by b_i. Therefore, given r speed scaling points, we partition the range $[1, W]$ (W is the WCEC of τ) into $r+1$ *phases*: Phase $0 = [b_0, b_1-1]$, Phase $1 = [b_1, b_2-1]$, ..., Phase $r = [b_r, b_{r+1}-1]$, where $b_0 = 1$ and $b_{r+1} = W+1$ are used to simplify the formulation.

Let us redefine *speed schedule* as the speeds of all phases. Our goal is to find a speed schedule that minimizes the expected energy consumption while still meeting the deadline D. Let the speed for Phase i ($0 \leq i \leq r$) be denoted by s_i, where $s_i \in \{f_1, \ldots, f_M\}$. Let the energy consumed by a single cycle in Phase i be denoted by $e(s_i)$, where $e(s_i) = \frac{\hat{p}(s_i)}{s_i}$ (recall from Section 3.3 that $\hat{p}(\cdot)$ does not account for the idle power). Thus, the energy consumed by Phase i is $F_i e(s_i)$, where $F_i = \sum_{b_i \leq j < b_{i+1}} (1 - cdf(j-1))$. Let $P_C(i) = 1 - cdf(b_i - 1)$, denoting the probability that the execution will reach b_i cycles and hence will go to Phase i. Assuming that the processor is idle and operating at the minimum frequency at the beginning of the frame, we obtain the following mathematical program:

$$\texttt{Minimize} \quad P_E(f_1, s_0) + F_0 e(s_0) + \sum_{1 \leq i \leq r} (P_C(i) P_E(s_{i-1}, s_i) + F_i e(s_i)) \tag{4.6}$$

$$\texttt{Subject to} \quad P_T(f_1, s_0) + \frac{w_0}{s_0} + \sum_{1 \leq i \leq r} \left(P_T(s_{i-1}, s_i) + \frac{w_i}{s_i} \right) \leq D \tag{4.7}$$

$$s_i \in \{f_1, f_2, \ldots, f_M\} \tag{4.8}$$

where $P_E(\cdot, \cdot)$ and $P_T(\cdot, \cdot)$ are the energy and time cost for speed changes (refer to Section 3.3.2) and $w_i = b_{i+1} - b_i$.

4.3.1.2 Patching the Schemes Obtained under the Ideal Processor Model

PACE[2] [40] and GRACE [71] proposed patching the schemes obtained under the ideal processor model. Both PACE and GRACE apply the well-known cubic-root rule of the power functions [14] and hence use $e(f) = c'_0 + c'_1 f^2$ (c'_0 and c'_1 are constants, f is the running speed) to approximate the actual energy/cycle function. Then they relax the constraint on s_i and assume that s_i is unrestricted and continuous. They also ignore the speed change overhead.

[2]We use the term *PACE* to refer to both DVS schemes under the ideal and realistic processor models proposed in [40].

This is equivalent to using the ideal processor model described in Section 3 in which α is set to 3. Thus, the minimization problem becomes

$$\texttt{Minimize} \quad c_0' D + \sum_{0 \le i \le r} F_i c_1' s_i^2 \tag{4.9}$$

$$\texttt{Subject to} \quad \sum_{0 \le i \le r} \frac{w_i}{s_i} \le D \tag{4.10}$$

$$0 \le s_i \le \infty \tag{4.11}$$

Notice that c_0' and c_1' have no effect on deciding the speed schedule. Using the Lagrange technique [36] or Jensen's inequality [35], the solution to (4.9)-(4.11) is

$$s_i = \frac{\sum_{0 \le j \le r} w_j F_j^{\frac{1}{3}}}{D F_i^{\frac{1}{3}}}$$

However, s_i needs to be changed to some available discrete frequency. This is where PACE and GRACE differ. GRACE is conservative in the sense that it rounds s_i up to the closest higher discrete frequency, whereas PACE rounds s_i to the closest discrete frequency (rounds up or down). Both schemes have shortcomings. For GRACE, s_i could be larger than the highest discrete frequency if F_i is very small. If this happens, s_i will have to be rounded down to the highest discrete frequency f_M, and therefore the deadline could be missed (considering that most of s_i's do not have this problem and they are rounded up, the probability of missing deadline should be reasonably small). For PACE, chances of missing the deadline are higher because PACE may round down. To solve this problem, PACE proposes to linearly scan all the phases to adjust the speeds [39]. GRACE does not deal with the speed change overhead, while PACE proposes to subtract the maximum possible time penalties from the allotted time. Appendix A uses an illustrative example to demonstrate the impact of speed rounding on the quality of the solution.

4.3.1.3 The PPACE Scheme We now present what we call the PPACE (Practical PACE) scheme under the realistic processor model. The heart of PPACE is a fully polynomial time approximation scheme (FPTAS) that can obtain ϵ-optimal solutions, which are within a factor of $1+\epsilon$ of the optimal solution and run in time polynomial in $1/\epsilon$. To better understand the problem expressed in (4.6)-(4.8), we give a graphical interpretation of the problem. First, we need the following definition:

Definition 1. *An energy-time label l is a 2-tuple (e,t), where e and t are nonnegative reals and denote energy and time respectively. We write the energy component as $l.e$ and the time component as $l.t$.*

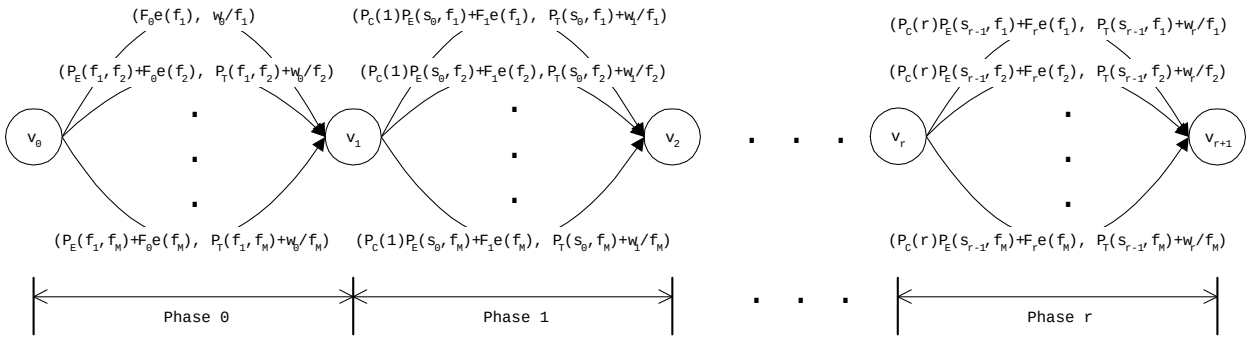


Figure 5: Graphical representation of the mathematical program (4.6)-(4.8)

The mathematical program (4.6)-(4.8) can be expressed as a graph $G=(V,E)$ shown in Figure 5. The vertex v_i $(1 \le i \le r+1)$ represents the point by which the first i phases have been already executed. The M edges between v_i and v_{i+1} $(0 \le i \le r)$ represent different speed choices (which frequency to use) for Phase i. Each choice is represented by an energy-time label, indicating the expected energy consumption and the worst-case running time for that phase. We also associate each path in the graph with an energy-time label, where the energy of a path is defined as the sum of the energies of all edges over the path and the time of a path is defined as the sum of the times of all edges over the path. When an energy-time label l is associated with a path, we denote the most recently used frequency by $l.f$. Therefore, the problem is reduced to finding a path from v_0 to v_{r+1} such that the energy

of the path is minimized while the time of the path is no greater than the deadline D.

Since a path can be summarized as an energy-time label, a straightforward approach is to start from v_0 and work all the way from left to right in an iterative manner to generate all paths. Each vertex v_i is associated with M energy-time label sets $LABEL(i, j)$, where $j = 1, 2, \ldots, M$. If a label $l \in LABEL(i, j)$, then we have $l.f = f_j$. For succinct presentation, we will use $LABEL(i, *)$ to denote all label sets in v_i (i.e., $LABEL(i, *)$ is shorthand for $\bigcup_{j=1}^{M} LABEL(i, j)$) and $LABEL(i, \cdot)$ to denote any label set in v_i (i.e., $LABEL(i, \cdot)$ is shorthand for $LABEL(i, j), 1 \leq j \leq M$). Initially all label sets are empty except for $LABEL(0, 1)$, which contains only one energy-time label $(0, 0)$. The whole process consists of $r+1$ iterations. In the i^{th} iteration, we generate all paths from v_0 to v_i: from $LABEL(i-1, *)$ we add the values of each edge, creating new paths, and store them in $LABEL(i, *)$. At the end, we just select the energy-time label with the minimum energy and with time no greater than D, from $LABEL(r + 1, *)$, as the solution to the problem.

Unfortunately, the size of $LABEL(i, \cdot)$ may suffer from exponential growth in this naive approach. To prevent this from happening, the key idea is to reduce and limit the size of $LABEL(i, \cdot)$ after each iteration by eliminating some of the energy-time labels in $LABEL(i, \cdot)$. We devise two types of eliminations: one that does not affect the optimality of the solution and one that may affect optimality but still allows for performance guarantee.

There are three eliminations that do not affect the optimality of the solution:

1. We can eliminate any energy-time label l in $LABEL(i, \cdot)$ if l cannot lead to a feasible solution (i.e., a solution that is guaranteed to meet the deadline but the resulting energy consumption is not necessarily optimized) by using a single frequency that is no less than the current frequency after Phase $(i - 1)$. This can be easily proved by contradiction. Suppose that l leads to a feasible solution l' for which the maximum frequency of Phase $i, i + 1, \ldots, r$ is f_{max} (f_{max} is not necessarily f_M because if the deadline is large enough, $f_{max} < f_M$ would be sufficient to meet the deadline). Then we can replace the frequencies of Phase $i, i + 1, \ldots, r$ with f_{max} and obtain another feasible solution, which leads to contradiction. Formally, the necessary condition for label l being able to lead to a

feasible solution is that there exists a frequency s, where $l.f \leq s \leq f_M$ such that

$$l.t + P_T(l.f, s) + \frac{\sum_{i \leq j \leq r} w_j}{s} \leq D \tag{4.12}$$

By manipulating Inequality (4.12) into a quadratic equation and solving it for s, we transform the necessary condition into

$$(D - l.t + \xi_1 l.f)^2 - 4\xi_1 \sum_{i \leq j \leq r} w_j \geq 0 \tag{4.13}$$

$$\frac{D - l.t + \xi_1 l.f - \sqrt{(D - l.t + \xi_1 l.f)^2 - 4\xi_1 \sum_{i \leq j \leq r} w_j}}{2\xi_1} \leq f_M \tag{4.14}$$

$$\frac{D - l.t + \xi_1 l.f + \sqrt{(D - l.t + \xi_1 l.f)^2 - 4\xi_1 \sum_{i \leq j \leq r} w_j}}{2\xi_1} \geq l.f \tag{4.15}$$

where ξ_1 is from (3.1).

2. For the second optimality-preserving elimination, we need to compare two energy-time labels.

   Definition 2. *Let $l_1 = (e_1, t_1)$ and $l_2 = (e_2, t_2)$ be two energy-time labels from $LABEL(i, \cdot)$. We say that l_1 dominates l_2, denoted by $l_2 \prec l_1$, if $e_1 \leq e_2$ and $t_1 \leq t_2$.*

   The dominance relation on a set of energy-time labels is clearly a partial ordering on the set. If $l_2 \prec l_1$, this means l_2 will not lead to any solution better than the best solution that l_1 leads to. Therefore, we eliminate all energy-time labels that are dominated by some other energy-time label in the same label set. Note that energy-time labels from different label sets cannot use Definition 5.2 to define dominance relation because of speed change overhead. This is also the reason why we have M label sets in each vertex. On the other hand, if speed change overhead can be ignored, we can combine M label sets into one.

   Performing the second elimination can be reduced to the maxima problem in computational geometry [52]. Specifically, the energy-time labels in label sets are stored in decreasing order of the energy component, breaking ties with smaller time component coming ahead. Thus, the elimination for a label set can be accomplished by using the dimension-sweep technique in time $O(n \ln n)$ [52], where n is the number of labels in the set.

3. For any energy-time label l in $LABEL(i, *)$ surviving the previous two eliminations, we compute a lower bound of the feasible solutions that l leads to (denoted by $l.LB$) and an upper bound of the best solution that l leads to (denoted by $l.UB$). Let $U = \min_{l' \in LABEL(i,*)} l'.UB$. Then U is an upper bound of the optimal solution. For an energy-time label l, if $l.LB > U$, that means l will not lead to the optimal solution and thus can be eliminated. A simple method to compute $l.LB$ for a label l is to compute the expected energy consumption assuming that the minimum frequency is used from Phase i on. Similarly, a simple method to compute $l.UB$ is to first find the optimal continuous frequency assuming that the task will run for the worst-case cycles, round it up and then compute the expected energy consumption assuming that this frequency is used from Phase i on. Note that these two methods only take constant time.

With the above eliminations, the size of $LABEL(i, \cdot)$ decreases substantially. Note that at this point the optimal solution is guaranteed to be found. However, the running time of the algorithm still has no polynomial time bound guarantee. Inspired by the fully polynomial time approximation scheme (FPTAS) of the subset-sum problem [20], we obtain a FPTAS for our problem, further reducing the size of $LABEL(i, \cdot)$.

The intuition for the FPTAS is that we need to further trim each $LABEL(i, \cdot)$ at the end of each iteration. A trimming parameter δ ($0 < \delta < 1$) will be used to direct the trimming. To trim an energy-time label set L by δ means to remove as many energy-time labels as possible, in such a way that if $\hat{L}$ is the result of trimming L, then for every energy-time label l that was removed from L, there is an energy-time label $\hat{l}$ in $\hat{L}$ such that $l.t > \hat{l}.t$ and $\frac{\hat{l}.e - l.e}{l.e} \leq \delta$ (or, equivalently, $\hat{l}.e \leq (1+\delta) l.e$). Such an $\hat{l}$ can be thought of as "representing" l in the new energy-time label set $\hat{L}$. Note that $\hat{L} \subseteq L$. Let the *performance guarantee* be ϵ ($0 < \epsilon < 1$), which means that the solution will be within a factor of $1 + \epsilon$ of the optimal solution. After the first type of eliminations (i.e., the optimality-preserving eliminations), $LABEL(i, \cdot)$ is trimmed using a parameter $\delta = (1+\epsilon)^{\frac{1}{r+1}} - 1$. The choice of δ shall be clear later in the proof of Theorem 2.

The procedure TRIM (shown in Algorithm 4.3) performs the second type of elimination for label set L. Note that the energy-time labels in label sets are stored in decreasing order on the energy component (or equivalently, in increasing order on the time component).

The PPACE scheme is shown in Algorithm 4.4.

Algorithm 4.3 TRIM($L = [l_1, l_2, \ldots, l_{|L|}]$,$\delta$)

1: $\hat{L} := \{l_1\}$
2: $last := l_1$
3: **for** $i := 2$ to $|L|$ **do**
4: **if** $last.e > (1+\delta)l_i.e$ **then**
5: append l_i onto the end of $\hat{L}$
6: $last := l_i$
7: **end if**
8: **end for**
9: **return** $\hat{L}$

4.3.1.4 Analysis of PPACE

We now show the time complexity of the procedure PPACE(ϵ) in Algorithm 4.4. First, notice that line 14 in Algorithm 4.4 corresponds to the TRIM procedure, that is, the optimality-preserving eliminations. Let $LABEL'(i, *)$ be the label sets obtained if lines 18-20 in Algorithm 4.4 are omitted. Note that $LABEL(i, j) \subseteq LABEL'(i, j)$, where $1 \leq j \leq M$ and the optimal solution is in $LABEL'(r + 1, *)$. By comparing $LABEL(i, *)$ and $LABEL'(i, *)$, we have the following lemma:

Lemma 2. *For every energy-time label $l' \in LABEL'(i, j)$, where $1 \leq j \leq M$, there exists a label $l \in LABEL(i, j)$ such that $l'.e \leq l.e \leq (1 + \delta)^i l'.e$ and $l'.t \geq l.t$.*

Lemma 2 shows how the error accumulates after each iteration when comparing label sets obtained with the second type of elimination and label sets obtained without the second type of elimination. The details of the proof is presented in Appendix C.

Theorem 2. *The procedure PPACE(ϵ) is a fully polynomial-time approximation scheme, that is, the solution that PPACE(ϵ) returns is within a factor of $1+\epsilon$ of the optimal solution and the running time is polynomial in $1/\epsilon$.*

Proof. Let l^* denote the optimal solution. Obviously $l^* \in LABEL'(r + 1, j)$, where $1 \leq j \leq M$. Then, by Lemma 2 there is a $l \in LABEL(r + 1, j)$ such that

$$l^*.e \leq l.e \leq (1 + \delta)^{r+1} l^*.e$$

Algorithm 4.4 PPACE(ϵ)

for $i := 1$ to $r+1$ **do**
  for $j := 1$ to M **do**
    $LABEL(i,j) := \phi$
  end for
end for
$LABEL(0,1) := \{(0,0)\}$
for $i := 1$ to $r+1$ **do**
  for each label $l \in LABEL(i-1,*)$ **do**
    for $j := 1$ to M **do**
      $LABEL(i,j) := LABEL(i,j) \cup (l.e + P_C(i-1)P_E(l.f, f_j) + F_{i-1}e(f_j), l.t + P_T(l.f, f_j) + \frac{w_{i-1}}{f_j}))$
    end for
  end for
  remove all $l \in LABEL(i,*)$ such that l does not satisfy (4.13)-(4.15)
  for $j := 1$ to M **do**
    remove all $l \in LABEL(i,j)$ such that $l \prec l'$, where $l' \neq l$ and $l' \in LABEL(i,j)$
  end for
  compute $l.LB$ and $l.UB$ for all $l \in LABEL(i,*)$ and then remove all $l \in LABEL(i,*)$ such that $l.LB > \min_{l' \in LABEL(i,*)} l'.UB$
  for $j := 1$ to M **do**
    $LABEL(i,j) :=$ TRIM($LABEL(i,j)$, $(1+\epsilon)^{\frac{1}{r+1}} - 1$)
  end for
end for
if no label in $LABEL(r+1,*)$ **then**
  return no solution
else
  return the label $l \in LABEL(r+1,*)$ with the minimum energy component
end if

Because we chose $\delta = (1+\epsilon)^{\frac{1}{r+1}} - 1$,

$$(1+\delta)^{r+1} = \left(1 + (1+\epsilon)^{\frac{1}{r+1}} - 1\right)^{r+1} = 1+\epsilon$$

then

$$l.e \leq (1+\epsilon) l^*.e$$

Therefore, the energy returned by PPACE(ϵ) is not greater than $1+\epsilon$ times the optimal solution, satisfying the first part of the theorem.

To show that its running time is polynomial in $1/\epsilon$, we first need to derive the upper bound on the size of $LABEL(i,j)$, where $1 \leq j \leq M$. Let $LABEL(i,j) = [l_1, l_2, \ldots, l_k]$ after trimming. We observe that the energies of any two successive energy-time labels differ by a factor of more than $(1+\delta)$ (otherwise, we would have already eliminated it). In particular,

$$l_1.e > (1+\delta) l_2.e > (1+\delta) 2 l_3.e \cdots > (1+\delta)^{k-1} l_k.e$$

Moreover, clearly $l_1.e \leq e(f_M) \sum_{0 \leq j \leq i} s_j F_j$ and $l_k.e \geq e(f_1) \sum_{0 \leq j \leq i} s_j F_j$. Let $\lambda = \frac{e(f_M)}{e(f_1)}$ (i.e., the ratio of the energies when running with the highest and lowest frequencies), then

$$(1+\delta)^{k-1} < \frac{l_1.e}{l_k.e} \leq \lambda$$

or, equivalently

$$k < 1 + \frac{\ln \lambda}{\ln(1+\delta)} = 1 + \frac{(r+1)\ln\lambda}{\ln(1+\epsilon)} = O(\frac{r \ln \lambda}{\epsilon}) \tag{4.16}$$

Now we can derive the running time of PPACE(ϵ) in Algorithm 4.4. There are $r+1$ iterations (line 7). In each iteration, the processing time is dominated by the dimension-sweep used for the second optimality-preserving elimination (line 15). Since the number of energy-time labels generated for each label set is $O(\frac{r \ln \lambda}{\epsilon})$, the processing time for each label set is $O(\frac{r \ln \lambda \ln(r \ln \lambda)}{\epsilon})$. Because there are M label sets in each vertex and there are $r+1$ iterations, the total running time is $O(\frac{r^2 M \ln \lambda \ln(r \ln \lambda)}{\epsilon})$. □

In fact, the running time of PPACE depends entirely on the total number of energy-time labels stored in all the vertices, which is $\sum_{i=0}^{r+1} \sum_{j=1}^{M} |LABEL(i,j)|$. The FPTAS is conservative, that is, the approximation guarantee reflects the performance of the algorithm only on the most pathological instances. In practice, using $\epsilon = 5\%$ usually gives a solution that is very close to the optimal solution (see experimental results in Section 4.3.4).

4.3.2 The Inter-task Scheme

For inter-task DVS under the realistic processor model, we propose a scheme called PITDVS (**P**ractical **I**nter-**I**ask **DVS**) that uses the optimal inter-task DVS scheme (OITDVS) from the ideal processor model as the basis and patches it to comply with the realistic processor model. Although PITDVS is not optimal, we hope that the advantage of using probabilistic information about the workload will outweigh the disadvantage of patching so that we can achieve better energy savings over the existing schemes.

Like OITDVS, the offline part of PITDVS is to compute the time allocation fraction β_i for each task τ_i. For the realistic processor model, a probability function is represented by a histogram. By treating the bins of a histogram as supercycles, it is trivial to transform the procedure OITDVS-offline in Algorithm 4.1 to become the offline part of PITDVS. The online part of PITDVS takes into account the issues of the realistic model; below we discuss these issues and provide corresponding solutions.

Patch 1 (Speed Change Overhead) Available processors have speed change overhead, including time penalty and energy penalty. When there are n tasks in the frame, the number of speed changes is at most n because the processor is expected to change speed only before the execution of each task. Also, the maximum time penalty is at most $P_T(f_1, f_M)$. Thus, we take a conservative approach. Before computing the speed for task τ_i to be executed, we subtract the maximum possible time penalty, $(N - i + 1)P_T(f_1, f_M)$ (recall that N is the number of tasks in the system), from the remaining time in the frame.

Patch 2 (Maximum and Minimum Speeds) The processor speed in the ideal processor model can be tuned from zero to infinity. In reality, however, every processor has a maximum speed f_M and a minimum speed f_1. The speed that is used to execute a task cannot violate these constraints, and thus we adjust the allotted time for τ_i at dispatch time as follows. Before starting to execute task τ_i and having time d left, if there is more time than needed to execute with the lowest speed ($\beta_i d > \frac{W_i}{f_1}$), then we allot time $\frac{W_i}{f_1}$ to τ_i (equivalent to using minimum speed f_1 to execute τ_i). Similarly, if there is less time than needed with

the maximum speed ($\beta_i d < \frac{W_i}{f_M}$), then we allot time $\frac{W_i}{f_M}$ to τ_i (equivalent to using maximum speed f_M to execute τ_i). Also, we need to compute the greedy-derived time t_{greedy}, that is, the conservative time obtained using Greedy scheme [48] to allow the rest of the tasks to finish by the deadline using the maximum speed. If $\beta_i d > \frac{W_i}{t_{greedy}}$, then we allot time $\frac{W_i}{t_{greedy}}$ to τ_i; this is because the Greedy scheme guarantees deadlines in the most aggressive form [48]. It is easy to see that the resulting schedule is still valid as long as the tasks can be scheduled using the maximum speed when all the tasks take their WCECs.

Patch 3 (Discrete Speeds) The processor speed in the ideal processor model can be tuned continuously. But real-world processors only provide a finite set of discrete speeds. Since we will use a constant speed to execute a task, the most straightforward way to fix this problem is to round the continuous speed up to the closest higher discrete speed [32].

To summarize the above patches, we show the online part of PITDVS in Algorithm 4.6, which calls Algorithm 4.5. The procedure PITDVS-online(i, d) is called before task τ_i is executed and there is time d remaining in the frame. This procedure can be regarded as running in constant time because the complexity of lines 2-4 in Algorithm 4.5 can be reduced to constant time by using an extra array to store the partial sums of W_i values. Thus, Patch 1 and 2 can be done in constant time. Patch 3 takes $O(\log M)$, where M is the number of available discrete speeds. Because M is usually small in practice, we can treat it as constant time.

We also propose a variant of the PITDVS scheme, PITDVS2, that uses up to two speeds to execute a task within the allotted time. This is due to the fact that any continuous speed can be emulated by using its two adjacent discrete speeds [32]. Intuitively, using two adjacent discrete speeds to emulate a continuous speed is expected to do better than rounding up the speed. For a continuous speed s, let the closest higher and lower discrete speeds be denoted by $\lceil s \rceil$ and $\lfloor s \rfloor$, respectively. To emulate s for time t, we let the system operate at $\lfloor s \rfloor$ for time t_1 and at $\lceil s \rceil$ for time $t - t_1 - P_T(\lfloor s \rfloor, \lceil s \rceil)$. Thus, we need to satisfy

$$\lfloor s \rfloor t_1 + \lceil s \rceil (t - t_1 - P_T(\lfloor s \rfloor, \lceil s \rceil) = s \cdot t \tag{4.17}$$

Algorithm 4.5 AdjustContinuousSpeed(i, d)

1: $W' := 0$ $\quad$ {W' is the remaining cycles after τ_i}
2: **for** $j := i + 1$ to N **do**
3: $\quad W' := W' + W_j$
4: **end for**
5: $d' := d - (N - i + 1)P_T(f_1, f_M)$ $\quad$ {Patch 1}
6: **if** $\frac{W_i + W'}{d'} \geq f_M$ **then**
7: $\quad$ **return** f_M
8: **end if**
9: $t := d'\beta_i$
10: {start of Patch 2}
11: **if** $t > \frac{W_i}{f_1}$ **then**
12: $\quad t := \frac{W_i}{f_1}$
13: **end if**
14: **if** $t < \frac{W_i}{f_M}$ **then**
15: $\quad t := \frac{W_i}{f_M}$
16: **end if**
17: $t_{greedy} := d' - W'/f_M$
18: **if** $t > t_{greedy}$ **then**
19: $\quad t := t_{greedy}$
20: **end if**
21: **return** $\frac{W_i}{t}$

Algorithm 4.6 PITDVS-online(i, d)

1: $s :=$ AdjustContinuousSpeed(i, d)
2: round s to the next available higher speed $\quad$ {Patch 3}
3: **return** s

Solving Equation (4.17) will give the speed schedule for PITDVS2, taking into account the speed change overhead. Algorithm 4.7 shows the online part of PITDVS2. Note that, although PITDVS2 changes speed during the execution of a task, we still categorize it as inter-task DVS scheme because its main idea is to emulate the optimal inter-task DVS scheme from the ideal processor model.

Algorithm 4.7 PITDVS2-online(i, d)

1: $s :=$ AdjustContinuousSpeed(i, d)
2: $t := \frac{W_i}{s}$
3: $s_1 :=$the closest discrete speed lower than s
4: $s_2 :=$the closest discrete speed higher than s
5: $t_1 := \frac{s_2(t-P_T(s_1,s_2))-s \cdot t}{s_2-s_1}$
6: $t_2 := t - t_1 - P_T(s_1, s_2)$
7: **return** $[[s_1, s_1t_1], [s_2, s_2t_2]]$

4.3.3 The Hybrid DVS Schemes

For hybrid DVS schemes under the realistic processor model, we can either patch the optimal hybrid DVS scheme obtained under the ideal processor model (GOPDVS), or combine the PITDVS and PPACE schemes, as described below.

The first new scheme is called PGOPDVS (Practical GOPDVS). The offline part of PGOPDVS assumes the ideal processor model. Its first step is to compute the time allocation fraction β_{ij} for each Phase j of task τ_i. By treating the phases of a task as supercycles, the first step can be done by the procedure GOPDVS-offline in Algorithm 4.2 with slight modification. The second step of the offline part of PGOPDVS is to compute the time allocation fractions $\hat{\beta}_i$ for the whole task τ_i. The latter can be computed from all the allocated times for each phase of task τ_i, as follows. When task τ_i is ready to execute and there is time d remaining in the frame, the time allocated for the first phase is $d_1 = \beta_{i1}d$; the time allocated for the second phase is $d_2 = (d - \beta_{i1}d)\beta_{i2} = (1 - \beta_{i1})\beta_{i2}d$. We can see that the time allocated to the j^{th} phase is $d_j = d\beta_{ij}\prod_{k=1}^{j-1}(1 - \beta_{ik})$. Thus, the time allocation fraction for the whole task τ_i

is

$$\hat{\beta}_i = \frac{\sum_{j=1}^{r_i} d_j}{d} = \sum_{j=1}^{r_i} \beta_{ij} \prod_{k=1}^{j-1} (1 - \beta_{ik})$$

where r_i is the number of phases of task τ_i. Once we have derived $\hat{\beta}_i$, we know the optimal (under the ideal processor model) time to be allocated to task τ_i before τ_i starts executing.

The online part of PGOPDVS performs the patches to fit the realistic processor model. The patches are similar to those of PITDVS described in Section 4.3.2. We first perform the same patches as Patch 1 and 2 of PITDVS. Then we follow the approach of PACE, that is, first round each speed to the closest available discrete speed, and then perform a linear scan and adjustment to make sure τ_i will finish in the allotted time in the worst case. Algorithm 4.8 shows the online part of PGOPDVS. The procedure PGOPDVS-online(i, d) is called before task τ_i is executed and there is time d remaining in the frame.

The second new scheme can be regarded as a variant of PITDVS. It differs from PITDVS only in that PPACE speed schedule is used to execute a task. Specifically, after the allotted time for a task is decided, procedure PPACE(ϵ) in Algorithm 4.4 is called to compute the speed schedule because different allotted times corresponds to different speed schedules. Thus we call this scheme PIT-PPACE. Algorithm 4.9 shows the online part of PIT-PPACE. Note that we are calling the offline part of the PPACE scheme (which has high time complexity) online in PIT-PPACE, which is by no means practical. This can be remedied by precomputing the PPACE speed schedules for all possible allotted time d for each task and storing them in memory for online lookup. We need to discretize the allotted time d because it is continuous. The degree of discretization, n_d, depends the characteristics of the task. In general, larger n_d results in better speed schedule at the expense of higher space overhead. The space overhead is proportional to n_d because for each task we need to precompute n_d speed schedules and store them for online use.

4.3.4 Evaluation

For the ideal processor model, we have described DVS schemes that are provably optimal for different DVS strategies. However, this is not the case for the realistic processor model. We have presented the PPACE scheme for frame-based systems with a single task, and the

Algorithm 4.8 PGOPDVS-online(i, d)

1: $W' := 0$ $\{W'$ is the remaining cycles after $\tau_i\}$
2: **for** $j := i + 1$ to N **do**
3: $\quad W' := W' + W_j$
4: **end for**
5: $d' := d - (N - i + 1)P_T(f_1, f_M)$ {Patch 1}
6: **if** $\frac{W_i + W'}{d'} \geq f_M$ **then**
7: $\quad$ **return** $[s_j = f_M], j = 1, 2, \ldots, r_i$
8: **end if**
9: $t := d'\hat{\beta}_i$
10: {start of Patch 2}
11: **if** $t > \frac{W_i}{f_1}$ **then**
12: $\quad t := \frac{W_i}{f_1}$
13: **end if**
14: **if** $t < \frac{W_i}{f_M}$ **then**
15: $\quad t := \frac{W_i}{f_M}$
16: **end if**
17: $t_{greedy} := d' - W'/f_M$
18: **if** $t > t_{greedy}$ **then**
19: $\quad t := t_{greedy}$
20: **end if**
21: **for** $j := 1$ to r_i **do**
22: $\quad s_j := t\beta_{ij}$
23: $\quad$ round s_j to the closest available discrete speed
24: **end for**
25: {linear scan}
26: $j := r_i$
27: **while** τ_i cannot finish by the deadline **do**
28: $\quad$ increase s_j to the next higher discrete speed
29: $\quad j := j - 1$
30: **end while**
31: **return** $[s_j], i = 1, 2, \ldots, r_i$

Algorithm 4.9 PIT-PPACE-online(i, d, ϵ)

1: $s :=$ AdjustContinuousSpeed(i, d)
2: $t := \frac{W_i}{s}$
3: set the deadline of τ_i to t
4: **return** PPACE(ϵ)

PITDVS, PITDVS2, PGOPDVS, and PIT-PPACE schemes for general frame-based systems. For these schemes, two major questions remain unanswered: (i) since the worst-case performance of PPACE is dependent on the value of ϵ, what is the appropriate ϵ value that should be used and how well does it perform compared to the existing schemes? (ii) because PITDVS, PITDVS2, PGOPDVS, and PIT-PPACE are approximations of the optimal schemes obtained under the ideal processor model and they do not have theoretical performance guarantees as PPACE does, how well do they perform in practice? To answer these questions, we conducted extensive simulations for different processor models and workloads described in Section 3.6.

4.3.4.1 Evaluation of Intra-Task DVS Schemes We used the six distributions (Figure 2) described in Section 3.6 to generate synthetic tasks. The power scaling factor is 1. The $WCEC$ is $500,000,000$ and the minimum number of cycles is $5,000,000$. This is corresponding to the ratio of the WCET to the best-case execution time being 100, as reported in [54]. The default number of speed scaling points is $r = 100$ and they are placed evenly in the range of $[1, WCEC]$. We also evaluate the effect of parameter r on the performance of the schemes.

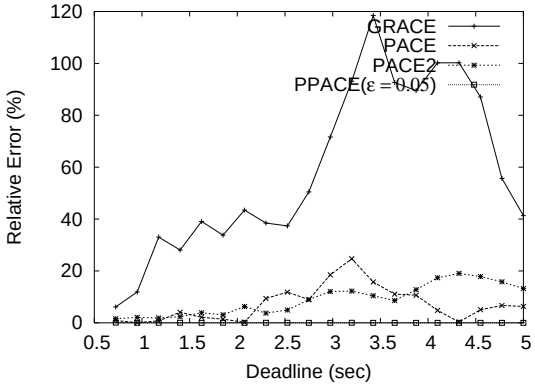
We define the relative error for any scheme that returns the expected energy consumption E to be $\frac{E-OPT}{OPT}$, where OPT is the optimal solution. We compute OPT using the PPACE scheme without doing the trimming operation at the expense of much longer running time. As usual with FPTAS algorithms, we set $\epsilon = 0.05$ for PPACE when comparing it with other schemes. For all experiments, we varied the slack available for power management. The slack is changed by varying the deadline from $\frac{WCEC}{f_M}$ to $\frac{WCEC}{f_1}$ (increasing the deadline will increase the slack, that is, will increase the allotted time for the task, thus resulting in less

energy consumption).

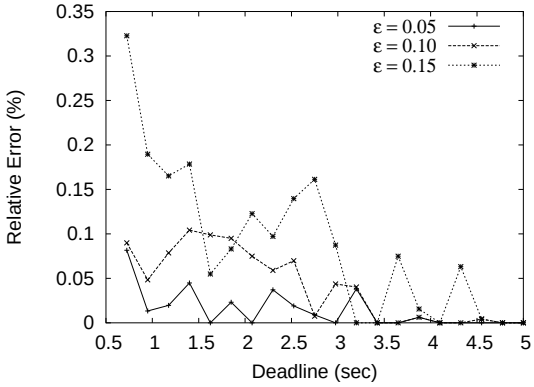
Under the above setup, we performed three experiments.

Experiment 1: Comparing All Schemes. In this experiment, we compare all intra-task DVS schemes including a variant of the PACE scheme called PACE2, which differs from PACE in that it uses two adjacent discrete speeds to emulate any continuous speed [32] obtained from the solution to the mathematical program (4.9)-(4.11). Since all schemes except for PPACE are based on the ideal processor model, which does not model speed change overheads, we assume that they fix this problem by subtracting the maximum possible time penalties from the allotted time. We only show the results for bimodal1 distribution (Figure 2(e)) because results for all other distributions are similar. As shown in Figure 6 (note the different Y-axes scales), PPACE is very close to optimal and outperforms all other schemes in all cases. On the other hand, the relative errors of GRACE, PACE, and PACE2 depend on the power model, the distribution of number of execution cycles, and the deadline. For the Synthetic processor (Figure 6(a)) for which the relative errors of GRACE, PACE and PACE2 are only due to rounding, PACE and PACE2 perform reasonably well, but still have relative errors up to 22% and 19% respectively. Since PACE2 uses two adjacent discrete speeds to emulate any continuous speed, it performs very well for processors that have good approximate analytical power functions (for XScale in Figure 6(c), PACE2's relative errors are less than 8%). However, PACE2 performs relatively poorly for the PowerPC 405LP, which does not have good approximate analytical power function (the relative error is up to 66%). Still, for XScale and PowerPC 405LP in Figure 6(e), neither PACE nor GRACE is a clear winner. In summary, the effect of approximations (i.e., using analytical functions to approximate the actual power function, and rounding or using two speeds to emulate any continuous speed) and patching for dealing with the speed change overhead could compound or cancel each other out. Thus, the solutions returned by GRACE, PACE, and PACE2 are unpredictable and unstable, while PPACE can provide performance guarantee.

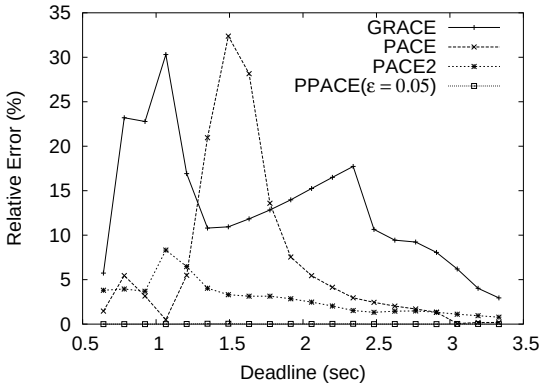
Figures 6(b), 6(d), and 6(f) show the results of sensitivity analysis on ϵ for PPACE. We can see that the relative errors are generally below 0.1%, 0.18%, 0.33% for $\epsilon = 5\%, 10\%, 15\%$, respectively. This means that, in practice, PPACE performs much better (around 2 orders



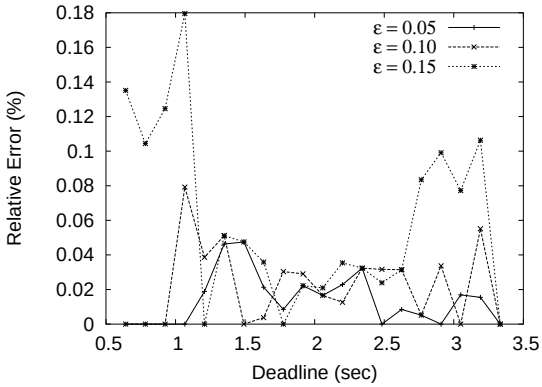
(a) Synthetic processor



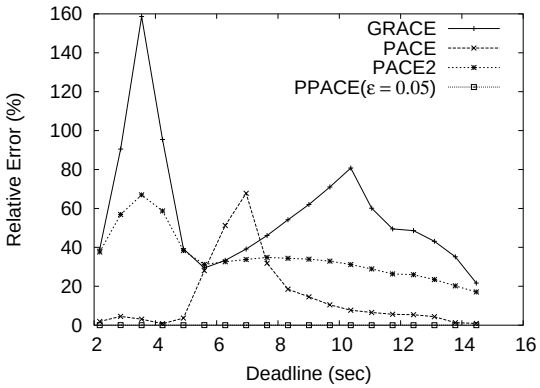
(b) Effect of ϵ on PPACE for Synthetic processor



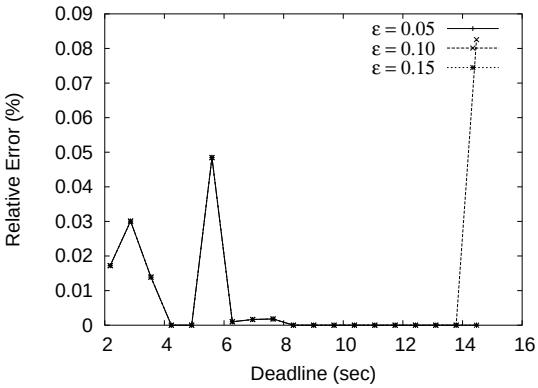
(c) XScale



(d) Effect of ϵ on PPACE for XScale



(e) PowerPC 405LP

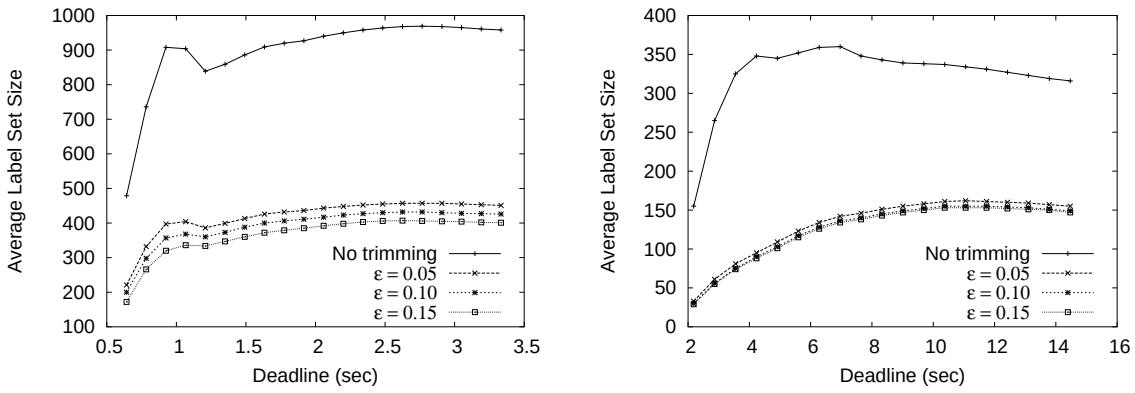


(f) Effect of ϵ on PPACE for PowerPC 405LP

Figure 6: Comparing intra-task DVS schemes for bimodal1 distribution (the relative errors are relative to optimal solutions)

of magnitude) than the performance guarantees it offers. This knowledge allows system designers to set the parameter ϵ higher than required, in order to speed up the algorithm execution, if the worst-case performance guarantee is not needed.

The time complexity of PPACE is greater than those of PACE and GRACE, which are $O(r \log M)$. In practice, the running time of PPACE depends on the implementation and the platform [3]. However, it is roughly proportional to the number of energy-time labels generated during the execution of PPACE(ϵ). Figure 7 compares the average number of energy-time labels in all vertices for different versions of PPACE. The curves on the top in Figure 7(a) and 7(b) are for PPACE without doing the trimming operation. We can see that the elimination of energy-time labels that affects optimality significantly reduces the size of the label set in each vertex but still allows for performance guarantee. It can also be observed from Figure 7 that the average number of energy-time labels increases when ϵ decreases, but they are all relatively small (especially for PowerPC 405LP shown in Figure 7(b)). It is also true that when M, the number of discrete speeds, increases, the average number of labels increases; this can be seen by comparing Figure 7(a) and 7(b).



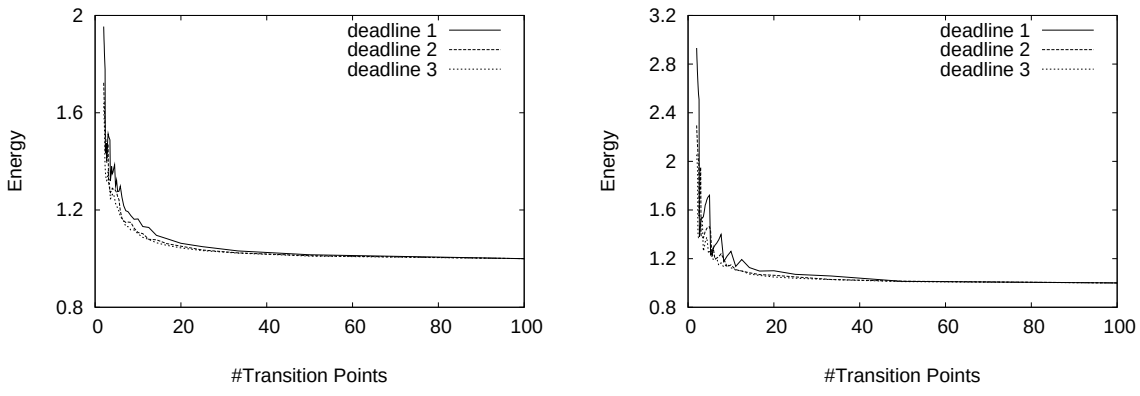
(a) XScale

(b) PowerPC 405LP

Figure 7: Efficiency of PPACE (bimodal1 distribution)

[3] In this work, we use Java to implement PPACE and our hardware setting is Pentium 4 3.2 GHz with 1 GB memory. The running time of PPACE(0.05) for all processor models is always within 1 second.

Experiment 2: Effect of Speed Scaling Points. Intra-task DVS schemes assume a predefined set of speed scaling points in tasks. However, it is not clear how to choose an optimal sequence of speed scaling points, especially in the presence of speed change overheads. This is still an open problem and it is beyond the scope of this dissertation. Intuitively, having more speed scaling points should result in a better speed schedule at the expense of longer time to find the speed schedule. Thus, we perform an experiment to find out how the number of speed scaling points affects the speed schedule. For the tasks in Experiment 1, we set the number of speed scaling points to be 2, 3, ..., 100 (the positions of the speed scaling points are evenly distributed) and obtained the speed schedule returned by PPACE with $\epsilon = 0.05$. Figure 8 shows the energy consumption (normalized to the energy consumption for the number of speed scaling points equal to 100) versus number of speed scaling points for three values of the deadline (for other values of the deadline, the curve is similar; also we do not show results for the Synthetic processor because they are similar). The results agree with our intuition. However, Figure 8 shows the phenomenon of diminishing return in increasing the number of speed scaling points. In practice, this fact will help system designer find a good number of speed scaling points more quickly; from our experiments, it seems that r=20 or r=30 is a good number, for these parameters.



(a) XScale

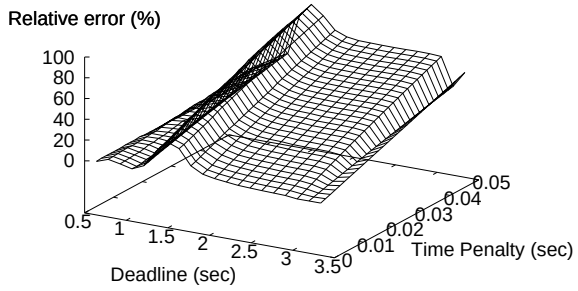
(b) PowerPC 405LP

Figure 8: Effect of speed scaling points (bimodal1 distribution)

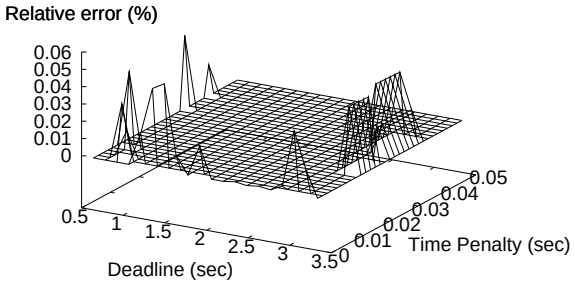
Experiment 3: Effect of Speed Change Overhead. We also performed simulations to evaluate the effect of speed change overhead. In practice, processors that can adjust the voltage internally have low overheads (in the range of microseconds), but systems that require changing the voltage externally experience high overheads in the milliseconds range. Using the same task sets used in Experiment 1, we varied the worst-case time penalty of XScale and PowerPC405LP for speed changes from 50μs to 50ms. The energy penalty is changed proportionally to the time penalty. For example, the energy penalty of XScale for time penalty being $t\mu$s is 1.2μJ $\times \frac{t}{12}$. Figure 9 shows the relative errors for PACE and PPACE (we do not show results for GRACE and PACE2 because similar conclusion can be reached). We can see that the relative errors for PPACE are all below 0.06%. For PACE, which has the smallest error, the relative error increases as the time penalty increases; when the slack is small and time penalty is large, the relative error can be up to 94%.

4.3.4.2 Evaluation of the DVS Schemes for General Frame-based Systems

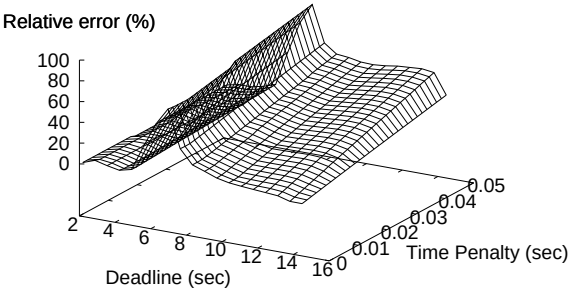
Evaluation on Synthetic Workloads. A frame-based real-time system is characterized by the number of tasks, the power scaling factor for each task, the WCEC of each task, the probability distribution of the number of execution cycles of each task, and the frame length. We simulated systems consisting of 5 and 10 tasks. We only show the results for the systems with 5 tasks because the results for systems with 10 tasks are similar. The power scaling factor was randomly chosen uniformly from 0.8 to 1.2. As with the simulations in Section 4.3.4.1, the WCEC of each task is 500,000,000 and the minimum number of cycles is 5,000,000. The probability function of each task's actual execution cycles is randomly chosen from the 6 representative distributions shown in Figure 2. The bin width of the histograms denoting the probability functions is 5,000,000 cycles. We experimented with 20 frame lengths chosen evenly from $\frac{5 \times WCEC}{f_M}$ to $\frac{5 \times WCEC}{f_1}$. For each simulated system, we evaluated 7 DVS schemes: Proportional2, Greedy2, Statistical2, PITDVS, PITDVS2, PGOPDVS and PIT-PPACE. The Proportional2, Greedy2, and Statistical2 schemes are extensions to the original Proportional, Greedy, and Statistical schemes [48] described in Section 2.4.1 by using up to two speeds to execute a task [32]. Note that these 7 schemes include inter-task and hybrid DVS schemes. We also evaluated a clairvoyant scheme, which is aware of the actual



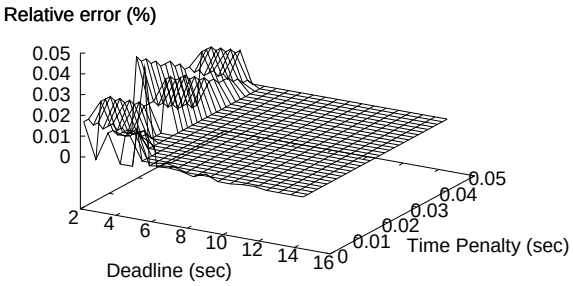
(a) PACE, XScale



(b) PPACE($\epsilon = 0.05$), XScale



(c) PACE, PowerPC 405LP



(d) PPACE($\epsilon = 0.05$), PowerPC 405LP

Figure 9: Effect of speed change overhead (bimodal1 distribution)

execution cycles of each task and uses the optimal frequency to execute each task. The clairvoyant scheme is used as the baseline to compare all other schemes.

In evaluating a DVS scheme on a system, we computed the average energy consumption per frame as the energy consumption for that scheme on that system. For the same system, we compute the relative error of a scheme whose energy consumption is E as $\frac{E-OPT}{OPT}$, where OPT is the energy consumption of the clairvoyant scheme. For each DVS scheme, we averaged the relative errors for all systems with the same frame length because we consider slack to be the most influential factor for energy consumption. Under the aforementioned setup, we simulated a total of 20 billion frames. The comparisons of the DVS schemes are shown in Figure 10.

For most of the simulations, the best scheme is either PITDVS2 or PIT-PPACE. For the simulations in which the best scheme is neither PITDVS2 nor PIT-PPACE, the minimum energy consumption of these two schemes is just off by less than 0.1% compared to the energy consumption of the best scheme. Thus, we take a closer look at PITDVS2 and PIT-PPACE on the plots in the right column of Figure 10. PITDVS2 performs no worse than PIT-PPACE in most cases except for a small range of frame lengths for PowerPC 405LP. Considering the high memory overhead or high run-time overhead of PIT-PPACE (as described in Section 4.3.3), we regard PITDVS2 as the better scheme.

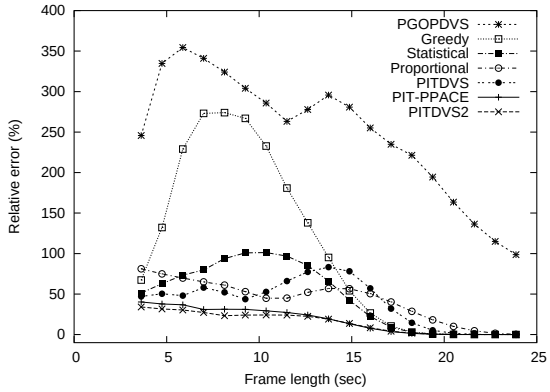
From Figure 10, we can see that for most cases the worst scheme is PGOPDVS, which is a surprising result considering that it is based on the best scheme under the ideal processor model. This is because the excessive rounding of speeds in the PGOPDVS scheme makes it drift far away from the optimal solution. The PITDVS scheme is not necessarily better than the DVS schemes that do not use probabilistic information of the workload. This is because the rounding-up effect offsets the advantage of using probabilistic information. Considering the difference between PITDVS and PITDVS2, we can see that using two adjacent discrete speeds to emulate a continuous speed [32] plays an important role in PITDVS2. Among the Proportional, Greedy, and Statistical schemes, none is a clear winner. This is because all of them are just based on heuristics and will only perform well in a subset of the problem space.

The two key factors that affect the energy savings of the PITDVS2 scheme over other

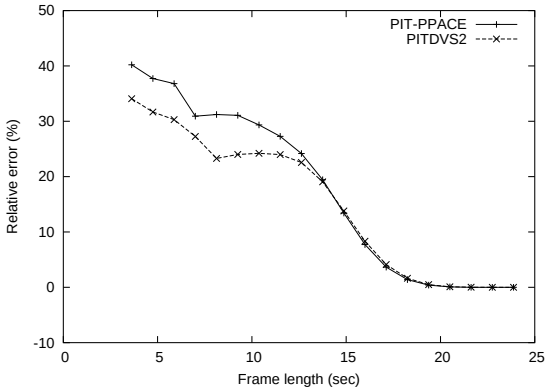
schemes (by comparing relative errors) are the minimum speed of the processor and the number of speeds available in the processor. In computing speed schedules, the PITDVS2 scheme is based on the solution under the ideal processor model in which the frequency of the processor is unrestricted and continuous. Because of the convexity of the power function, high speeds are not usually obtained by the PITDVS2 scheme. But low speeds are desired because the PITDVS2 scheme can navigate the full spectrum of available speeds and can find the best speed that minimizes the expected energy consumption. The importance of the number of speeds available in the processor is obvious given that we need to convert the continuous speeds to discrete speeds. For example, because the minimum speed of the Synthetic processor is less than that of XScale, and the number of speeds of that processor is greater than that of XScale, the energy saving for the Synthetic processor is greater than that for XScale.

We also performed simulations to evaluate the effect of speed change overhead. We varied the time penalty of XScale and PowerPC 405LP in the same way as in Section 4.3.4.1. The results are very similar to those in Figure 10. This is because the derivations of all schemes are based on the ideal processor model and they all deal with the speed change overhead in a similar way. That is, they ignore the energy penalty and subtract the maximum possible time penalty from the available time when a task is being scheduled to execute. As the time penalty increases, the energy consumption of all schemes increases, but their differences in terms of energy consumption remain roughly the same when averaging the results.

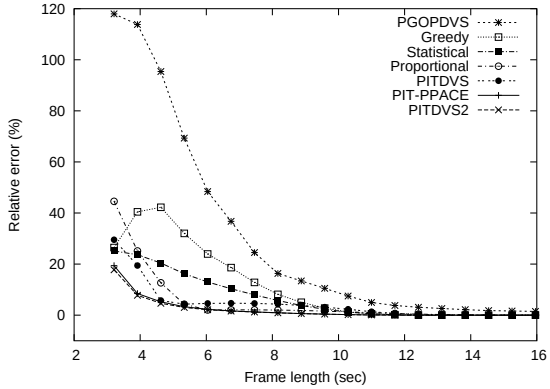
Evaluation on Real-World Workload. We evaluated the DVS schemes on automatic target recognition (ATR) described in Section 3.6. An example embedded system that uses ATR is an unmanned autonomous vehicle (UAV) with two cameras installed. Each camera will take 1 to 3 pictures every 100 ms and send them to a back-end for target recognition. The back-end is required to finish processing each batch of pictures in a timely fashion. Thus, the back-end can be modeled as a frame-based system whose frame length is 100 ms. A task in this system is responsible for processing a picture. For any given frame, there could be 2 to 6 tasks, each corresponding to a picture. The number of execution cycles of a task depends on the number of ROIs in the picture that the task is processing. The number



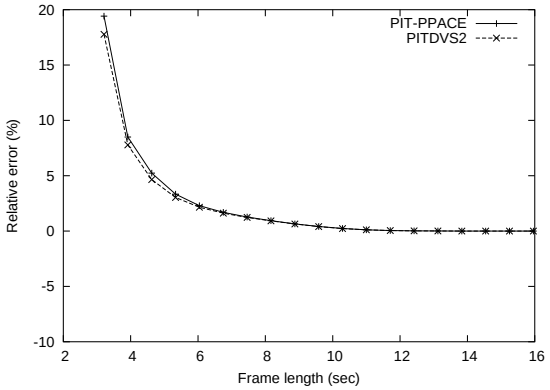
(a) Synthetic processor, all schemes



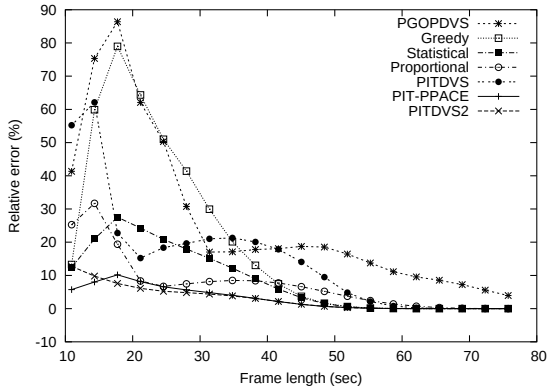
(b) Synthetic processor, 2 schemes



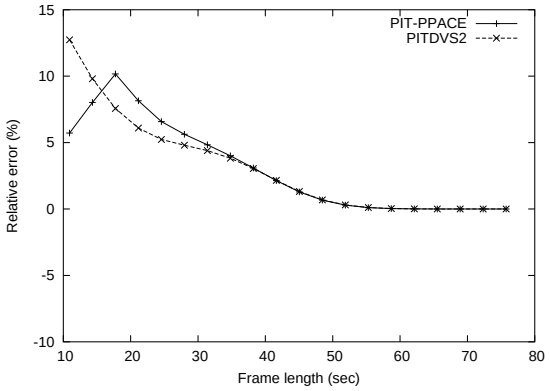
(c) XScale, all schemes



(d) XScale, 2 schemes



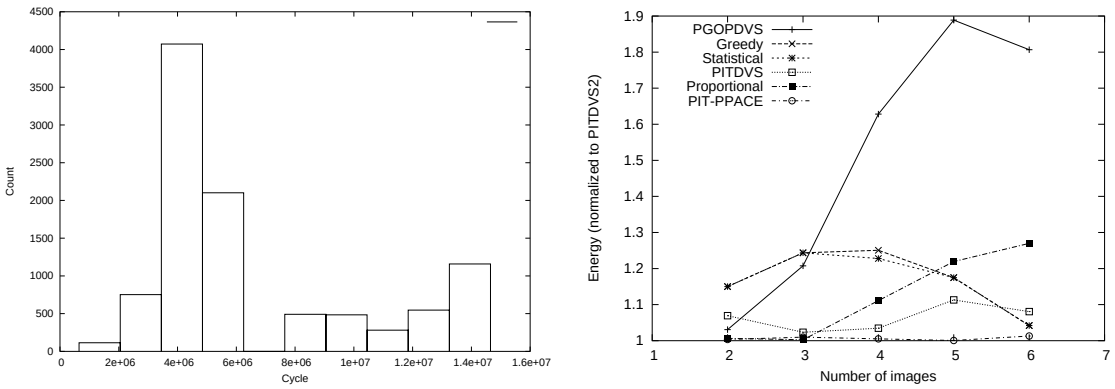
(e) PowerPC 405LP, all schemes



(f) PowerPC 405LP, 2 schemes

Figure 10: Comparison of DVS schemes for general frame-based systems (the relative errors are relative to the clairvoyant scheme)

of ROIs in a picture cannot be predicted before processing the picture. We assume that the back-end is equipped with an Intel XScale processor.



(a) Histogram of execution cycles for ATR

(b) Comparison of DVS schemes

Figure 11: Experimental results for ATR

We obtained the probability distribution of cycle demand of the task by profiling on a training image set using Simplescalar [6]. Figure 11(a) shows the histogram of the execution cycles. We then precomputed the schedule for having 2, 3, 4, 5, and 6 images to be processed in one frame. The five schedules are stored in the back-end. When a period begins, the back-end counts the number of images received and applies the corresponding schedule. Note that fewer number of images corresponds to more slack in the frame. Figure 11(b) shows the energy consumption of each scheme *normalized to that of the PITDVS2 scheme* when the back-end has 2, 3, 4, 5, and 6 images to process. From the figure we can see that the PITDVS2 scheme can achieve significant energy savings over the previously existing schemes. For example, the PITDVS2 scheme can achieve an average of 11% energy saving over the Proportional scheme.

4.4 A UNIFIED APPROACH

In the previous section, we have investigated DVS schemes under the realistic processor model. For inter-task and hybrid DVS schemes, we patched the optimal DVS schemes under the ideal model (e.g., rounding continuous speed to available discrete speed) in order to comply with the realistic model. As a result, the optimal speeds that were derived based on the ideal model are no longer optimal for the realistic model.

Experiments in Section 4.3.4 show some anomaly for the patched DVS schemes based on the ideal model. For example, the best of all DVS schemes for the ideal model is the optimal stochastic hybrid DVS scheme called GOPDVS [72] (refer to Section 4.2.3). However, we have seen in Section 4.3.4 that the patched GOPDVS performs even worse than certain schemes that do not use any stochastic information of workloads (e.g., the Proportional scheme). This is discouraging since using more information is supposed to lead to better results. Even for the stochastic schemes that were shown experimentally to outperform non-stochastic schemes, it is not clear how well those stochastic schemes perform when compared to the optimal stochastic scheme under the realistic model, which is yet to be found.

In this section, we provide a step function based approach for obtaining the optimal stochastic DVS schemes under the realistic model. To control the computational complexity, we use a function approximation technique to obtain DVS schemes whose resulting expected energy consumption is guaranteed to be within a factor of $1 + \epsilon$ of the optimal solution and whose time complexity is polynomial in $\frac{1}{\epsilon}$, where ϵ is a parameter of the DVS schemes. As with PPACE, our approximation technique falls in the category of fully polynomial time approximation schemes (FPTAS) [20]. Our approach is *unified* in the sense that it can be used to obtain all three types of DVS schemes (i.e., inter-task DVS, intra-task DVS, and hybrid DVS).

4.4.1 Problem Formulation

To facilitate the presentation, we formally define DVS schemes. A DVS scheme consists of N *speed schedule* functions $S_i(\cdot)$ $(i = 1, 2, \ldots, N)$. $S_i(t)$ denotes the speed schedule for task

τ_i, when τ_i is ready to execute and there is time t remaining in the frame. A speed schedule for a task dictates what speed(s) to be used for executing this task. For inter-task DVS, a speed schedule is a single speed; for intra-task and hybrid DVS, each speed schedule contains a set of speeds and the corresponding speed scaling points.

Let $e'_i(\varsigma, x)$ and $t'_i(\varsigma, x)$ denote the energy consumption and time for executing τ_i using speed schedule ς when the actual number of execution cycles of τ_i is x. The expected energy consumption for executing $\tau_i, \tau_{i+1}, \ldots, \tau_N$ using time t can be computed recursively as

$$E_i(t) = \sum_{k=1}^{r_i} P_i(k)\big(e'_i(S_i(t), B_i(k)) + E_{i+1}(t - t'_i(S_i(t), B_i(k)))\big)$$

and $E_{N+1}(t) = 0$. Thus, the goal is to find DVS schemes that minimize $E_1(D)$.

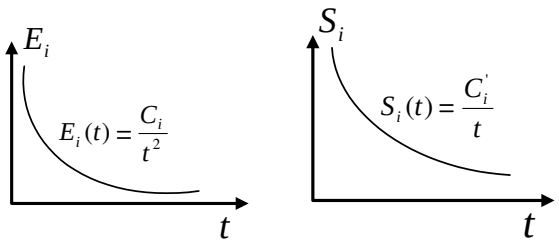
We present three DVS schemes in this section: (1) SIDVS, which stands for the Simple Inter-task DVS scheme that employs inter-task DVS strategy in the absence of speed change overhead; (2) IDVS, which is a generalization of the SIDVS scheme that considers speed change overhead; (3) HDVS, which stands for hybrid DVS schemes (also considering speed change overhead). Obviously, SIDVS is the simplest scheme. The differences between SIDVS under the realistic model and that under the ideal model are discrete speeds vs. continuous speeds, and arbitrary power function vs. well-defined power function. We present the SIDVS scheme because it derivation contains all the essential ingredients of our approach and can be easily extended to form the other schemes.

4.4.2 The Basic Idea for the Unified Approach

In this section, we describe the basic idea behind our approach through the discussion of the main idea of the SIDVS scheme. The purpose of this section is to illustrate all the key elements in our approach without delving into too much mathematical detail. We start by describing the SIDVS scheme, followed by the properties of this scheme and how to obtain such a scheme.

4.4.2.1 The SIDVS Scheme The SIDVS scheme (simple inter-task DVS) contains $2N$ functions, two for each task in the system. Specifically, each task τ_i corresponds to two functions: $E_i(\cdot)$ and $S_i(\cdot)$. These two functions denote that when task τ_i is ready to execute and there is time t remaining in the frame, if we use speed $S_i(t)$ to execute τ_i, the minimum expected energy consumption, $E_i(t)$, of executing $\tau_i, \tau_{i+1}, \ldots, \tau_N$ will be achieved. Computing functions $E_i(\cdot)$ and $S_i(\cdot)$ (i.e., finding all the mappings in the functions) is done offline, which we will discuss in Section 4.4.2.3. During the operation of the system, the OS scheduler will consult functions $S_i(\cdot)$ to determine the speed of each task. Specifically, at the beginning of a frame when there is time D available, the OS scheduler will use the speed $S_1(D)$ to execute τ_1. After τ_1 finishes and it has taken time t', there is time $D - t'$ remaining in the frame and the OS scheduler will use the speed $S_2(D - t')$ to execute τ_2. The same process will be applied to the rest of the tasks.

4.4.2.2 Properties of the SIDVS Scheme Before discussing how to obtain the SIDVS scheme, we examine the properties of functions $E_i(\cdot)$ and $S_i(\cdot)$, which will determine their representation. During the examination, we also consider the scheme under the ideal model, which will help us understand the motivation behind our approach.

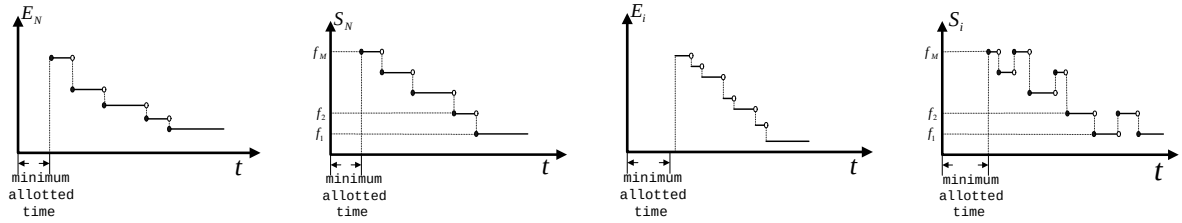


(a) $E_i(\cdot)$ $(1 \leq i \leq N)$ (b) $S_i(\cdot)$ $(1 \leq i \leq N)$

Figure 12: The SIDVS Scheme for the ideal model

We first examine functions $E_N(\cdot)$ and $S_N(\cdot)$ because they only involve a single task τ_N. For the ideal model (assuming cubic power/frequency relationship), we have $E_N(t) = \frac{C_N}{t^2}$ and $S_N(t) = \frac{C'_N}{t}$, where neither C_N nor C'_N depends on t. Thus, both $E_N(\cdot)$ and $S_N(\cdot)$ (see

Figure 12) can be represented by just a constant (C_N for $E_N(\cdot)$ and C'_N for $S_N(\cdot)$). This is due to the simplicity of the ideal model.



(a) $E_N(\cdot)$ (b) $S_N(\cdot)$ (c) $E_i(\cdot)$, $i < N$ (d) $S_i(\cdot)$, $i < N$

Figure 13: The SIDVS Scheme for the realistic model

For the realistic model, however, both $E_N(\cdot)$ and $S_N(\cdot)$ are step functions (piece-wise constant functions). Figure 13(b) shows the function of $S_N(\cdot)$ for the realistic model, which can be obtained by rounding its counterpart for the ideal model (Figure 12(b)) to the available discrete speeds. We can see that there are M (M is the number of available discrete speeds) *half-open* line segments in the graph, each corresponding to an available discrete speed. Each line segment can be represented by its left end point because its right end point is the left end point of the line segment to its immediate right or infinity when it is the rightmost line segment of the graph. We call the left end point of a line segment a *turning point*. Thus, $S_N(\cdot)$ can be represented by $2M$ numbers because each turning point can be represented by its two coordinates. Figure 13(a) shows the function of $E_N(\cdot)$, which has also M half-open line segments. Counting from left to right, the k^{th} ($1 \leq k \leq M$) line segment of $E_N(\cdot)$ shares the same starting t coordinate and ending t coordinate with the k^{th} line segment of $S_N(\cdot)$. Thus, $E_N(\cdot)$ can be also represented by $2M$ numbers as in $S_N(\cdot)$. Note that $E_N(\cdot)$ does not include the idle energy consumption, as explained in Section 3.3. Computing $E_N(t)$ and $S_N(t)$ can be turned into a table lookup, which takes $O(\log M)$ if binary search is used.

Now we examine functions $E_i(\cdot)$ and $S_i(\cdot)$ ($1 \leq i < N$), which involve multiple tasks. For the ideal model, $E_i(\cdot)$ ($1 \leq i < N$) is of the same form as $E_N(\cdot)$, that is, $E_i(t) = \frac{C_i}{t^2}$, where C_i does not depend on t. The same holds for $S_i(\cdot)$ ($1 \leq i < N$). This elegant result, which was proved in Section 4.2.2, is again due to the simplicity of the ideal model. Thus,

both $E_i(\cdot)$ and $S_i(\cdot)$ ($1 \leq i < N$, see Figure 12) can still be represented by a single constant. This means that the complexity of the representation for the ideal model does not depend on i. However, this is not the case for the realistic model.

For the realistic model, both $E_i(\cdot)$ and $S_i(\cdot)$ ($1 \leq i < N$, see Figures 13(c) and 13(d)) are still step functions. But there are more turning points in $E_i(\cdot)$ ($1 \leq i < N$) than in $E_N(\cdot)$. This is because $E_i(\cdot)$ is the expected energy consumption for multiple tasks and different combination of speeds from these tasks usually results in different energy consumption. In fact, the number of turning points of $E_i(\cdot)$ may suffer from exponential growth as i decreases, which will be clear in Section 4.4.3.2. As in the case for $E_N(\cdot)$ and $S_N(\cdot)$, each line segment of $E_i(\cdot)$ can be translated into one in $S_i(\cdot)$. However, the number of possible values of $S_i(\cdot)$ is only M. If two adjacent line segments in $S_i(\cdot)$ share the same S_i coordinate, they can be combined into one line segment. Thus, the number of turning points of $S_i(\cdot)$ is usually much smaller than that of $E_i(\cdot)$. As for the shape of the function, $E_i(\cdot)$ (Figure 13(c)) is still an non-increasing function, while $S_i(t)$ (Figure 13(d)) may go up and down as t increases.

The latter claim is counter-intuitive, especially for the speed going up when t increases (i.e., if there is more slack, the speed increases to yield lower energy consumption). This is due to the nature of discrete speeds. We describe a scenario where this will happen. Suppose that for some workload it is beneficial to use low speed to execute the tasks following τ_i. For a given available time t, increasing the speed for τ_i will not give enough room to drop the speed for the following tasks to the next lower discrete speed. However, as the available time t increases, increasing the speed for τ_i will eventually be rewarded.

Similar to computing $E_N(t)$ or $S_N(t)$, computing $E_i(t)$ or $S_i(t)$ ($1 \leq i < N$) is a table lookup, which takes $O(\log K)$, where K is the number of turning points in the function.

4.4.2.3 Obtaining the SIDVS Scheme

From Section 4.4.2.2, we can see that computing $E_i(\cdot)$ and $S_i(\cdot)$ is equivalent to identifying all the turning points in $E_i(\cdot)$ and $S_i(\cdot)$. From the recursive description of the problem in Section 4.4.1 it is natural to compute $E_i(\cdot)$ and $S_i(\cdot)$ in reverse order, that is, first compute $E_N(\cdot)$ and $S_N(\cdot)$, then $E_{N-1}(\cdot)$ and $S_{N-1}(\cdot)$, and so on. The computation of $E_i(\cdot)$ and $S_i(\cdot)$ only depends on $E_{i+1}(\cdot)$, as $E_{i+1}(\cdot)$ has already "summarized" functions $E_j(\cdot)$ and $S_j(\cdot)$, where $j = i+2, \ldots, N$. When the computation

is done, all $E_i(\cdot)$ $(i = 1, 2, \ldots, N)$ can be discarded because they are not needed for the operation of the system.

As mentioned in Section 4.4.2.2, the number of turning points in the functions may suffer exponential growth. Thus, we propose a function approximation technique to limit the number of points. We use an example to illustrate the technique. Consider the two turning points, (e_1, t_1) and (e_2, t_2), inside the circle in Figure 14. Obviously, $e_1 > e_2$ and $t_1 < t_2$. If the difference between e_1 and e_2 is small (formally, if $\frac{e_1 - e_2}{e_2} < \delta$, where δ is a parameter to quantify the difference), we eliminate the point (e_2, t_2). Through this kind of elimination, the number of turning points is reduced and upper bounded by a polynomial in $\frac{1}{\delta}$. However, the resulting function is only an approximation of the original function (i.e., the elimination induces error). This is because when time t, where $t_2 \leq t < t_3$, is available, we cannot use the speed schedule corresponding to (e_2, t_2) since it was eliminated. We will have to use the speed schedule corresponding to (e_1, t_1) and thus result in expected energy e_1 that is greater than e_2. Because of the way we eliminate the points, the difference between the resulting function and the original function is guaranteed to be no more than a factor δ of the original function.

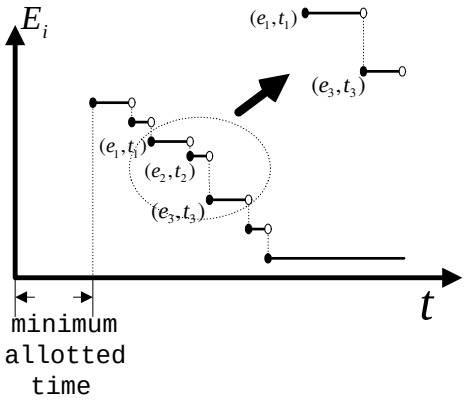


Figure 14: Function approximation

We apply the above function approximation technique to function $E_i(\cdot)$ before computing $E_{i-1}(\cdot)$. Thus, the error accumulates and increases as i decreases. Let the error of $E_1(\cdot)$ be denoted by ϵ. The expected energy consumption of the system, $E_1(D)$, is within a factor of

$1+\epsilon$ of the optimal expected energy consumption. If we let ϵ be a parameter set by system designers, we can derive the value of δ to be used for each function approximation. The technical detail can be found in Section 4.4.3.2.

4.4.3 The Details of the Unified Approach

Having described the basic idea behind our approach in the previous section, we provide the technical details in this section. We first formally define step function and related operations in Section 4.4.3.1. Then we give the algorithm to obtain the SIDVS scheme and present its analysis in Section 4.4.3.2. Finally, we extend the algorithm for the SIDVS scheme to obtain the IDVS and HDVS schemes in Section 4.4.3.3 and 4.4.3.4, respectively.

4.4.3.1 On Step Functions From Section 4.4.2 we can see that step functions play an important role in our approach. Thus, being able to represent step functions effectively and manipulate step functions efficiently are crucial for the viability of our approach.

We first formally define step function through the following two definitions.

Definition 3. *A point $\mathbb{P}$ is a 2-tuple (e,t), where e and t are nonnegative reals and denote energy and time respectively. We write the energy component as $\mathbb{P}.e$ and the time component as $\mathbb{P}.t$.*

Definition 4. *A step function (piece-wise constant function) $\mathbb{F}(\cdot)$ is defined as a point sequence $\mathbb{S}=[\mathbb{P}_1,\mathbb{P}_2,\ldots,\mathbb{P}_m]$, where $\mathbb{P}_1.t<\mathbb{P}_2.t<\cdots<\mathbb{P}_m.t$. $\mathbb{F}(t)$ is undefined when $t<\mathbb{P}_1.t$, otherwise $\mathbb{F}(t)=\mathbb{P}_i.e$ and $i=\max_{j=1,2,\ldots,m}\{j|t\geq\mathbb{P}_j.t\}$. The cardinality of $\mathbb{S}$ is also called the size of function $\mathbb{F}(\cdot)$.*

Having formally defined step function, we will use $\mathbb{F}$ to denote a step function $\mathbb{F}(\cdot)$ unless confusion arises. Let $|\mathbb{F}|$ denote the number of points in $\mathbb{F}$. Obviously, computing $\mathbb{F}(t)$ can be done in time $O(\log|\mathbb{F}|)$ by using binary search.

We now look at three operations between a number and a step function.

Definition 5. *The operator $+_e$ is defined between a real x and a step function $\mathbb{F}$ such that $x+_e\mathbb{F}=[(x+\mathbb{P}_1.e,\mathbb{P}_1.t),(x+\mathbb{P}_2.e,\mathbb{P}_2.t),\ldots]$ (i.e., the result is still a step function). Other operators, $\times_e$ and $+_t$ can be defined similarly.*

Obviously, the operators defined in Definition 5 can be performed in time $O(|\mathbb{F}|)$.

Finally, we describe two operations between step functions.

Definition 6. *The sum operator* $+_{\mathbb{F}}$ *is defined between two step functions,* $\mathbb{F}_1$ *and* $\mathbb{F}_2$*, such that* $\mathbb{F}_1 +_{\mathbb{F}} \mathbb{F}_2 = \mathbb{F}$ *and* $\mathbb{F}(t) = \mathbb{F}_1(t) + \mathbb{F}_2(t)$*. The merge operator* $\cup$ *is defined between two step functions,* $\mathbb{F}_1$ *and* $\mathbb{F}_2$ *such that* $\mathbb{F}_1 \cup \mathbb{F}_2 = \mathbb{F}$ *and* $\mathbb{F}(t) = min\,(\mathbb{F}_1(t), \mathbb{F}_2(t))$*.*

The resulting step function $\mathbb{F}$ by either the sum or the merge operators over n step functions $\mathbb{F}_i$ $(i = 1, 2, \ldots, n)$ could have as many as $\sum_{i=1}^{n} |\mathbb{F}_i|$ points. The time component of each point in $\mathbb{F}$ comes from one of the $\mathbb{F}_i$'s. Because the points in $\mathbb{F}_i$ are already sorted, the time components of all points in $\mathbb{F}$ can be obtained by a procedure similar to merge sort in time $O((\sum_{i=1}^{n} |\mathbb{F}_i|) \log n)$. To compute the energy component of each point in $\mathbb{F}$, the sum operator takes constant time and the merge operator takes $O(\log n)$ time by using a priority queue. Thus, computing $+_{\mathbb{F}}{}_{i=1}^{n}\mathbb{F}_i$ takes $O((\sum_{i=1}^{n} |\mathbb{F}_i|) \log n)$ time and computing $\cup_{i=1}^{n}\mathbb{F}_i$ takes $O((\sum_{i=1}^{n} |\mathbb{F}_i|) \log^2 n)$ time.

4.4.3.2 The Algorithm for SIDVS

Recall from Section 4.4.2.3 that we compute functions E_i and S_i in reverse order. For succinct presentation, we do not show the computation of functions S_i because it can be easily performed as a by-product of computing E_i.

To compute function E_i, we first consider M helper functions $\hat{E}_{i,j}$ $(j = 1, 2, \ldots, M)$, where M is the number of available discrete speeds. $\hat{E}_{i,j}$ denotes the expected energy function when frequency f_j is used to execute task τ_i. A single value of $\hat{E}_{i,j}$ can be computed as

$$\begin{aligned} \hat{E}_{i,j}(t) &= \sum_{k=1}^{r_i} P_i(k) \left(\hat{p}_i(f_j) \frac{B_i(k)}{f_j} + E_{i+1}(t - \frac{B_i(k)}{f_j}) \right) \\ &= \hat{p}_i(f_j) \frac{A_i}{f_j} + \sum_{k=1}^{r_i} P_i(k) E_{i+1}(t - \frac{B_i(k)}{f_j}) \end{aligned}$$

In the above equations, $\hat{p}_i(f_j)\frac{B_i(k)}{f_j}$ is the energy consumption of executing the first k bins of τ_i and $E_{i+1}(t - \frac{B_i(k)}{f_j})$ is the expected energy consumption of executing $\tau_{i+1}, \ldots, \tau_N$. Computing the whole function $\hat{E}_{i,j}$ can be expressed using our notations about step functions as Line 6 in Algorithm 4.10. Function E_i is just the result of merging M $\hat{E}_{i,j}$ functions. During the merging process, the optimal speed corresponding to each point is also determined. The

optimal algorithm to solve SIDVS is shown at Lines 1-8 in Algorithm 4.10. Line 9 is used for the function approximation technique mentioned in Section 4.4.2.3 and will be explained at the end of this section.

We now analyze the time complexity and space complexity of computing E_i. The key operation in computing $\hat{E}_{i,j}$ is the sum operation over r_i step functions, each is of size $|E_{i+1}|$. Thus, the time to compute $\hat{E}_{i,j}$ is $O(r_i|E_{i+1}|\log r_i)$ and the number of points in $\hat{E}_{i,j}$ is $O(r_i|E_{i+1}|)$. The key operation in computing E_i is the merge operation over M step functions, each is of size $O(r_i|E_{i+1}|)$. Thus, the time to compute E_i is $O(Mr_i|E_{i+1}|\log^2 M)$ and the number of points in E_i is $O(Mr_i|E_{i+1}|)$. Since the base case is $|E_{N+1}| = 1$, we can obtain the closed forms of the time complexity and space complexity to be $O((Mr_i)^{N-i+1}\log^2 M)$ and $O((Mr_i)^{N-i+1})$, respectively.

Algorithm 4.10 SIDVS(ϵ)

1: $E_{N+1} := \{(0,0)\}$
2: **for** $i := N$ downto 1 **do**
3: {compute E_i}
4: **for** $j := 1$ to M **do**
5: {the case where f_j is used to execute τ_i}
6: $\hat{E}_{i,j} := \hat{p}_i(f_j)\frac{A_i}{f_j} +_e +_{\mathbb{F}_{k=1}^{r_i}} P_i(k) \times_e \left(\frac{B_i(k)}{f_j} +_t E_{i+1}\right)$
7: **end for**
8: $E_i := \cup_{j=1}^{M} \hat{E}_{i,j}$
9: $E_i := TRIM(E_i, (1+\epsilon)^{\frac{1}{N}} - 1)$
10: **end for**

The time complexity of the optimal algorithm for the SIDVS scheme depends greatly on the size of functions E_i. As we can see from the analysis of the optimal algorithm, the size of E_i may grow exponentially as i goes from N to 1. Thus, we need to control the size of function E_i within some polynomial bound. To do that, we trim (i.e., remove some points) function E_i after it is computed at Line 7 in Algorithm 4.10. A trimming parameter δ ($0 < \delta < 1$) is used to direct the trimming. After function E_i is trimmed, the energy components of any adjacent points (recall from Definition 4 that the points are stored in the order of increasing t component) differ by at least a factor of δ. The choice of $\delta = (1+\epsilon)^{\frac{1}{N}} - 1$

(ϵ is a parameter of the SIDVS scheme) at Line 9 in Algorithm 4.10 will be clear at the end of this section. Algorithm 4.11 shows the trimming procedure.

Algorithm 4.11 TRIM($\mathbb{F} = [\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_{|\mathbb{P}|}]$,$\delta$)

1: $\hat{\mathbb{F}} := \{\mathbb{P}_1\}$
2: $l := \mathbb{P}_1$
3: **for** $i := 2$ to $|\mathbb{F}|$ **do**
4: **if** $l.e > (1+\delta)\mathbb{P}_i.e$ **then**
5: append $\mathbb{P}_i$ onto the end of $\hat{\mathbb{F}}$
6: $l := \mathbb{P}_i$
7: **end if**
8: **end for**
9: **return** $\hat{\mathbb{F}}$

The function approximation achieved by the trimming procedure is inspired by [20] and is similar to the label elimination technique used in Section 4.3.1. Thus, we only sketch its analysis for the sake of completeness

Before computing the number of points in E_i after trimming, we prove an important lemma. Let E'_i ($i = 1, 2, \ldots, N$) be the step functions obtained if Line 9 in Algorithm 4.10 is omitted. That is, E'_i is the set of functions returned by the optimal algorithm. By comparing E'_i and E_i, we have the following lemma:

Lemma 3. *For every point $\mathbb{P}' \in E'_i$ where $1 \leq i \leq N+1$, there exists a point $\mathbb{P} \in E_i$ such that $\mathbb{P}'.e \leq \mathbb{P}.e \leq (1+\delta)^{N+1-i}\mathbb{P}'.e$ and $\mathbb{P}'.t \geq \mathbb{P}.t$.*

Proof. This lemma is equivalent to $E_i(t) \leq (1+\delta)^{N+1-i}E'_i(t)$ for any value of t. The proof is by induction on i and the base case for $i = N+1$ obviously holds from Line 1 in Algorithm 4.10. In the induction step for E_i, we inspect Line 6 in Algorithm 4.10. From the hypothesis, $E_{i+1}(t)$ is within a factor of of $(1+\delta)^{N-i}$ of $E'_{i+1}(t)$. All the operations at Line 6 will preserve this property. After the trimming operation, the factor will be only increased by $(1+\delta)$, which will make $E_i(t)$ with a factor of $(1+\delta)^{N-i+1}$ of $E'_i(t)$. □

Using functions E'_i will lead to expected energy consumption of $E'_1(D)$ and using functions E_i will lead to expected energy consumption of $E_1(D)$. From Lemma 3, we have $E_1(D) \leq$

$(1+\delta)^N E'_1(D)$. Since we choose δ to be $(1+\epsilon)^{\frac{1}{N}} - 1$, we have $E_1(D) \leq (1+\epsilon)E'_1(D)$.

To compute the upper bound of the number of points in E_i, we note that after the trimming procedure, the energy components of any adjacent points differ by at least a factor of δ. Let the leftmost point in E_i be denoted by $\mathbb{P}_l$ (which is upper bounded by the energy consumption when all tasks use the maximum speed) and the rightmost point in E_i be denoted by $\mathbb{P}_r$ (which is lower bounded by the energy consumption when all tasks use the minimum speed). Thus, we have

$$\mathbb{P}_l.e > (1+\delta)^{|E_i|-1}\mathbb{P}_r.e$$

By plugging in $\delta = (1+\epsilon)^{\frac{1}{N}} - 1$ and some algebraic manipulations, we will obtain $|E_i| = O(\frac{N \log \lambda}{\epsilon})$, where $\lambda = \frac{\mathbb{P}_l.e}{\mathbb{P}_r.e}$. Thus, the number of points in E_i is upper bounded by a polynomial in $\frac{1}{\epsilon}$.

4.4.3.3 The Algorithm for IDVS Recall from Section 4.4.1 that the IDVS scheme is a generalization of the SIDVS scheme that considers speed change overhead. The SIDVS scheme can be easily extended to form the IDVS scheme. Instead of computing only one expected energy function E_i for each task τ_i as in SIDVS, we compute M expected energy functions $E_{i,s}$ $(s = 1, 2, \ldots, M)$. $E_{i,s}$ denotes the expected energy consumption of executing tasks $\tau_i, \tau_{i+1}, \ldots, \tau_N$ when the current speed is f_s, that is, when the speed before the execution of τ_i starts is f_s. Computing each $E_{i,s}$ in IDVS is similar to computing E_i in SIDVS. The only difference is that computing $E_{i,s}$ takes into consideration the energy penalty and time penalty associated with the speed change. Thus, computing each $E_{i,s}$ in IDVS has the same time and space complexity as computing E_i in SIDVS. Algorithm 4.12 shows the details of the IDVS scheme.

In the IDVS scheme, $N \times M$ speed schedule functions are computed. During the operation of the system, when task τ_i is ready to execute and there is time t remaining in the frame, the OS scheduler will detect the current speed s of the processor and use speed $S_{i,s}(t)$ to execute τ_i.

Algorithm 4.12 IDVS(ϵ)

1: **for** $s := 1$ to M **do**
2: $E_{N+1,s} := \{(0,0)\}$
3: **end for**
4: **for** $i := N$ downto 1 **do**
5: **for** $s := 1$ to M **do**
6: {compute $E_{i,s}$}
7: **for** $j := 1$ to M **do**
8: {the case where f_j is used to execute τ_i}
9: $\hat{E}_{i,s,j} := P_E(f_s, f_j) + \hat{p}_i(f_j)\frac{A_i}{f_j} +_e +_{\mathbb{F}}{}_{k=1}^{r_i} P_i(k) \times_e \left(\frac{B_i(k)}{f_j} + P_T(f_s, f_j) +_t E_{i+1,j}\right)$
10: **end for**
11: $E_{i,s} := \cup_{j=1}^{M} \hat{E}_{i,s,j}$
12: $E_{i,s} := TRIM(E_{i,s}, (1+\epsilon)^{\frac{1}{N}} - 1)$
13: **end for**
14: **end for**

4.4.3.4 The Algorithm for HDVS In the HDVS (Hybrid DVS) scheme, a task is allowed to change speed during its execution. Because of the speed change overhead, a limited number of speed scaling points at which speed may change are predefined for a task [40]. This is similar to predefining the quantum size for OS due to the context switch overhead. The speed remains constant between any two adjacent speed scaling points of a task. For ease of presentation, we choose the bin boundary of the histogram representing the probability distribution of a task as the speed scaling points for the task. By treating each bin of a task as a subtask, the IDVS scheme can be easily extended to form the HDVS scheme. We add one more dimension to the expected energy functions for each task τ_i. That is, we compute function $E_{i,s,b}$ ($s = 1, 2, \ldots, M$ and $b = 1, 2, \ldots, r_i$) that denotes the expected energy consumption of executing bin $b, b+1, \ldots, r_i$ of task τ_i, and tasks $\tau_{i+1}, \ldots, \tau_N$ when the current speed is f_s. There is a catch, however. $E_{i,s,b}$ is not only dependent on $E_{i,\cdot,b+1}$, but also $E_{i+1,\cdot,1}$. This is because task τ_i may finish at bin b and the rest of the bins will not be executed. Let X be the number of cycles that τ_i executes. We compute $\hat{P}_i(b)$, the

probability that bin b of task τ_i will be executed given that the previous $b-1$ bins have been executed

$$\begin{aligned}\hat{P}_i(b) &= Prob(X \geq B_i(b) | X \geq B_i(b-1)) \\ &= \frac{Prob(X \geq B_i(b) \wedge X \geq B_i(b-1)}{Prob(X \geq B_i(b-1))} \\ &= \frac{1 - cdf_i(b-1)}{1 - cdf_i(b-2)}\end{aligned}$$

where $cdf_i(0) = cdf_i(-1) = 0$. Similar to the IDVS scheme, the helper function $\hat{E}_{i,s,b,j}$ denotes the expected energy consumption of executing bin $b, b+1, \ldots, r_i$ of task τ_i, and tasks $\tau_{i+1}, \ldots, \tau_N$ when the current speed is f_s and speed f_j is used to execute bin b. Thus, a single value of $\hat{E}_{i,s,b,j}$ can be computed as

$$\begin{aligned}\hat{E}_{i,s,b,j}(t) &= \hat{P}_i(b)(P_E(f_s, f_j) + \hat{p}_(f_j)\frac{w_i(b)}{f_j} + E_{i,j,b+1}(t \\ &\quad -P_T(f_s, f_j) - \frac{w_i(b)}{f_j})) + (1 - \hat{P}_i(b))E_{i+1,s,1}\end{aligned}$$

where $w_i(b) = B_i(b) - B_i(b-1)$. Algorithm 4.13 shows the details of the HDVS scheme.

In the HDVS scheme, the speed schedule functions for each bin of a task is computed. That is, there are a total of $M \times \sum_{i=1}^{N} r_i$ speed schedule functions. However, we do not need these many speed schedule functions for the operation of the system. This is because for a single task, the execution speed is always non-decreasing [40]. This indicates that the speed schedule of a task for a given amount of time has at most M speeds and M speed scaling points. Thus, we compute new speed schedule functions $\hat{S}_{i,s}$ (the number of speed schedules in $\hat{S}_{i,s}$ is $|S_{i,s,1}|$), which denote the speed schedules for the whole task τ_i when the current speed is f_s, from $S_{i,\cdot,b}$ $(b = 1, 2, \ldots, r_i)$ and use them during the operation of the system.

Note that when there is only one task in the system, the HDVS scheme essentially becomes an intra-task DVS scheme. In this case, the HDVS scheme is very similar to the PPACE scheme from user's standpoint. This is because both schemes are fully polynomial time approximation schemes, that is, both schemes can give performance guarantees and achieve very close to optimal solution. Their main difference is that PPACE computes the speed schedule from the first bin of the task to the last bin, while HDVS does in the opposite

Algorithm 4.13 HDVS(ϵ)

for $s := 1$ to M **do**
  $E_{N+1,s,1} := \{(0,0)\}$
end for
for $i := N$ downto 1 **do**
  for $b := r_i$ downto 1 **do**
    for $s := 1$ to M **do**
      {compute $E_{i,s,b}$}
      for $j := 1$ to M **do**
        {the case where f_j is used to run bin b of τ_i}
        $\hat{E}_{i,s,b,j} := \hat{P}_i(b) \times_e (P_E(f_s, f_j) + \hat{p}_i(f_j)\frac{w_i(b)}{f_j} +_e (P_T(f_s, f_j) + \frac{w_i(b)}{f_j} +_t E_{i,j,b+1})) + (1 - \hat{P}_i(b)) \times_e E_{i+1,s,1}$
      end for
      $E_{i,s,b} := \cup_{j=1}^{M} \hat{E}_{i,s,b,j}$
      $E_{i,s,b} := TRIM(E_{i,s,b}, (1+\epsilon)^{\frac{1}{\sum_{k=1}^{N} r_k}} - 1))$
    end for
  end for
end for

order. In fact, our experiments show that the performance of the PPACE scheme is almost identical to that of the HDVS scheme for one task in the sense that when setting ϵ to 5%, the relative errors compared to the optimal solution for both schemes are all below 0.1%.

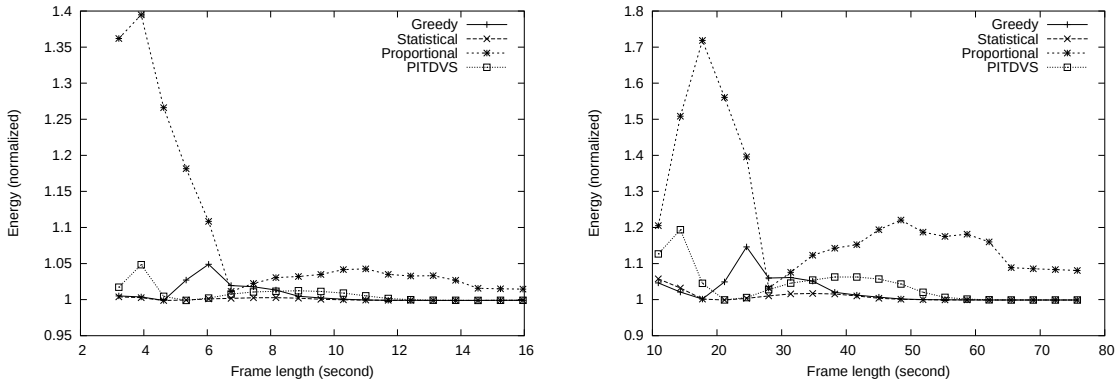
4.4.4 Evaluation Results

In this section, we use the IDVS and HDVS schemes as the baselines to experimentally evaluate the existing inter-task and hybrid DVS schemes, respectively, for general frame-based systems. All existing DVS schemes can be regarded as heuristic solutions because they do not have any performance guarantee under the realistic model. However, history has shown that for certain hard problems, there exist heuristic solutions that work very well and even close to the optimal in practice. The purpose of the evaluation in this section is to identify those good heuristic schemes. Note that in general hybrid schemes are better than inter-task DVS schemes. However, inter-task DVS schemes are easier to implement than hybrid schemes and sometimes are preferred by system designers. Thus, identifying good inter-task DVS schemes is also important.

The simulation setup is the same as that for evaluation of DVS schemes for general frame-based systems described in Section 4.3.4.2.

4.4.4.1 Evaluation of Inter-task DVS Schemes We evaluated four inter-task DVS schemes: Proportional, Greedy, Statistical, and PITDVS. The first three schemes [48] are non-stochastic schemes that do not use stochastic information of the workloads. They are all based on the ideal model and need to be patched to fit the realistic model. The PITDVS scheme is obtained by patching the optimal stochastic inter-task DVS scheme under the ideal model (refer to Section 4.3.4). The patches for all schemes are similar, including rounding continuous speed up to the lowest feasible discrete speed (i.e., guaranteed to meet deadlines) and subtracting the maximum possible time penalty from the available system time. The energy consumption of all schemes is normalized to that of the IDVS scheme with $\epsilon = 0.05$ (i.e., the energy consumption is guaranteed to be within 5% of the optimal). For all experiments, the number of points in $S_{i,s}$ of the IDVS scheme is at most 97. Recall from

Section 4.4.2.3 that we only need functions $S_{i,s}$ during the operation of the system. Thus, the space overhead of the IDVS scheme is very small.



(a) XScale

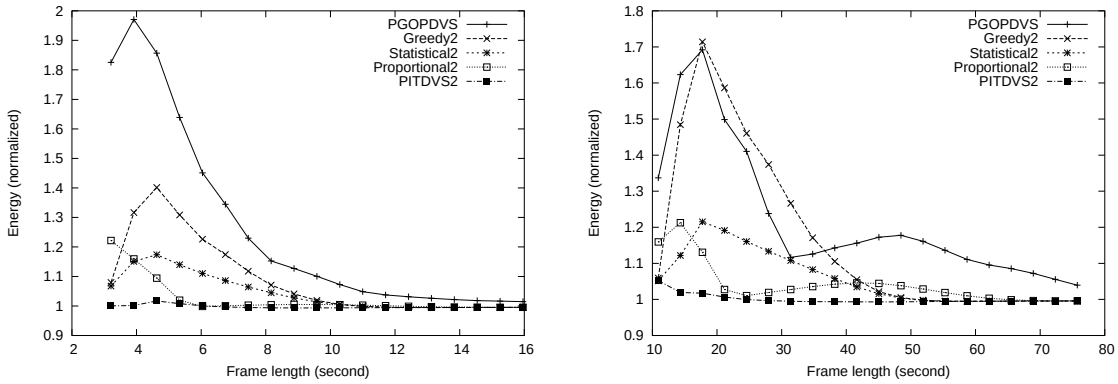
(b) PowerPC 405LP

Figure 15: Evaluation of Inter-task DVS Schemes (normalized to IDVS)

Figure 15 shows the evaluation results. We can see that the Statistical scheme is very close to IDVS, which is guaranteed to be within (1+5%) of the optimal and many times is in practice better than the guarantee. Thus, we conclude that Statistical is very close to the optimal even though it is not provably optimal for either the ideal or realistic model. The Greedy scheme also performs relatively well in most cases, comparing with the IDVS scheme. PITDVS is outperformed by the Statistical scheme in most cases. This is more evident for the PowerPC 405LP model. Even the Greedy scheme beats PITDVS in some cases. This is anomaly because the rounding-up effect offsets the advantage of using stochastic information.

4.4.4.2 Evaluation of Hybrid DVS Schemes We evaluated five hybrid DVS schemes: Proportional2, Greedy2, Statistical2, PITDVS2, and PGOPDVS. The first four schemes are different from their inter-task DVS counterparts only in that up to two speeds are used to execute a task. This is due to the fact that any continuous speed can be emulated by using its two adjacent discrete speeds [32]. In essence, these schemes attempt to emulate inter-task DVS schemes under the ideal model. However, they belong to hybrid DVS schemes

technically because the speed may change during the execution of a task (thus they need interrupt support). The PGOPDVS scheme is obtained by patching the optimal stochastic hybrid DVS scheme GOPDVS [72] under the ideal model (refer to Section 4.3.3 for the patching). The energy consumption of all schemes is normalized to that of the HDVS scheme with $\epsilon = 0.05$. For all experiments, the number of speed schedules in $\hat{S}_{i,s}$ of the HDVS scheme is at most 1013. Thus, the space overhead of the HDVS scheme is reasonably small.



(a) XScale

(b) PowerPC 405LP

Figure 16: Evaluation of hybrid DVS Schemes (normalized to HDVS)

Figure 16 shows the evaluation results. We note several quantitative differences from the results for inter-task DVS schemes. First, GOPDVS performs poorly, which is a surprising result considering that it is based on the best of all schemes under the ideal model. This is because the excessive rounding of speeds in the GOPDVS scheme makes it drift far away from the optimal solution. Statistical2 performs much worse than its inter-task DVS counterpart, when compared to other schemes. The Greedy2 scheme is the worst of all non-stochastic DVS schemes. The good performance of the PITDVS2 scheme is also a surprising result since its main idea is to emulate only the optimal stochastic inter-task DVS scheme under the ideal model.

4.5 SUMMARY

In this chapter, we investigated energy-aware uniprocessor scheduling problems for streaming applications. Since the problems related to deterministic workloads have been well studied, we focused on problems of scheduling stochastic workloads, namely, STREAM-UP-S-ST and STREAM-UP-S-TG. Solving these two problems is equivalent to finding DVS schemes for frame-based hard real-time systems that execute stochastic workloads.

We started out by investigating DVS schemes under the ideal processor model. Because of the simplicity of the model, provably optimal DVS schemes can be obtained and the optimal schemes for different DVS strategies share great similarities. Our main contributions are to propose OITDVS, the Optimal Inter-Task DVS scheme under the ideal processor model, and to provide a unified view of the optimal DVS schemes for all DVS strategies under the ideal processor model.

We then turned to DVS schemes for the realistic processor model. We presented PPACE (Practical PACE), which is a new DVS scheme for frame-based systems with a single task that takes into consideration discrete speeds and speed change overhead. PPACE can give performance guarantees and achieve energy savings very close to the optimal solution. For frame-based systems with multiple tasks, we proposed PITDVS2 (Practical Inter-Task DVS, using 2 speeds) and showed that it outperforms the existing DVS schemes in our experiments. We also showed that simple patches to optimal DVS schemes obtained under the ideal processor model do not necessarily generate DVS schemes that perform well in practice.

However, in investigating DVS schemes for the realistic model, we used different approaches for different DVS strategies. Furthermore, the PITDVS2 scheme is based on heuristics. Although it performs well experimentally, it is not clear that there still exists better schemes. Driven by this motivation, we proposed a unified approach for obtaining optimal (or provably close to optimal) stochastic inter-task, intra-task, and hybrid DVS schemes under the realistic processor model. As a result, optimal DVS schemes for all DVS strategies under the realistic model also share great similarity, as in the case for the ideal model. We used the optimal schemes to establish tight upper bounds on energy savings for stochastic DVS schemes and were able to identify good DVS schemes that are based on heuristics.

5.0 SCHEDULING IN MULTIPROCESSOR SYSTEMS

5.1 OVERVIEW

In this chapter, we consider energy-aware multiprocessor scheduling problems for streaming applications, that is, the STREAM-MP-D-ST, STREAM-MP-D-TG, STREAM-MP-S-ST, and STREAM-MP-S-TG problems that are described in Section 3.5.

As chip multiprocessors (CMPs) are quickly becoming the dominant computer architecture, scheduling streaming applications on multiprocessor systems has become increasingly important. CMP is the main solution to continuing improving computing performance beyond Moore's law. However, as the number of cores on a chip increases, so does the power density, making power management a major concern for CMPs. Compared to energy-aware scheduling of a streaming application on uniprocessor systems, multiprocessor scheduling poses more challenges, as follows.

1. **Static-dynamic power trade-off:** there is a fundamental trade-off between static and dynamic power consumption for multiprocessor systems. Assuming perfect parallelism for a given workload, as the number of active processors on a system increases, the static power consumption of the system increases, while the dynamic power consumption decreases since the load on each processor is smaller. As long as neither dynamic nor static power accounts for most of the total power, which is true for current technology, the two power management mechanisms, DVS and on-off, must be combined to optimize the power consumption that leads to minimum energy consumption for applications. Thus, we need to decide the number of active processors to execute a streaming application in addition to determining the execution speed of each task.

2. **Task mapping:** we need to decide the mapping of tasks to active processors in multiprocessor scheduling. In general, task mapping for multiprocessor systems without consideration of energy is already a hard problem [25]. Thus, energy-aware task mapping will only add more complexity. Even for streaming applications that have only singleton task graph representation, multiprocessor systems open up the opportunity to execute different instances of a task on multiple processors and we need to decide the number of instances in order to optimize the energy consumption.
3. **Two QoS requirements:** the QoS constraints, throughput and response time, can no longer be collapsed into one constraint in multiprocessor scheduling. Energy-aware scheduling of streaming applications on multiprocessor systems, assuming that the number of active processors is given and the deadline is equal to the period, have been extensively studied (for example, [5]). However, scheduling under both throughput and response time constraints deserves research attention for the following reasons. First, the period is shorter than the deadline in many situations, as for example when applying automatic target recognition (ATR) in unmanned autonomous vehicle (UAV). Scheduling algorithms that assume equal deadline and period force streaming applications to service requests faster than required, which goes against the common DVS wisdom of slowing down processors for just-in-time completion. Second, finding the appropriate number of active processors to execute an application is crucial for saving energy. We will show that high static power may force streaming applications into servicing requests faster than required (in spite of period being less than deadline) in order to save energy, which is counterintuitive for DVS. Third, even when the optimal number of active processors is known, the task mapping and task speed scheduling (i.e., deciding task speeds) become more complex for multiprocessor scheduling problems because of the interplay of the two QoS requirements.

In this chapter, we investigate energy-aware scheduling algorithms for multiprocessor systems. The outcome of the scheduling algorithms are: (i) the number of active processors to execute the task graph; (ii) the mapping of tasks to active processors; and (iii) the execution speed of each task (also called *speed schedule*). As in the case for uniprocessor systems, the ultimate goal is to obtain scheduling algorithms under the realistic processor

model. Thus, we only use the ideal processor model as a stepping stone in the investigation and do not necessarily devise a scheduling algorithm under the ideal processor model for each problem under investigation.

Table 6 shows the road map of our investigation in this chapter. We first consider scheduling a single task on multiprocessor systems. Our solution is a master-slave scheme that executes different instances of the task on multiple processors to satisfy the QoS requirements while trying to minimize the energy consumption. The proposed schemes for deterministic and stochastic workloads are presented in Section 5.2.1 and 5.2.2, respectively. We then investigate how to schedule a task graph on multiprocessor systems. For deterministic workload. A fully polynomial time approximation scheme called Scheduling1D is proposed for scheduling linear task graphs in Section 5.3.1. In Section 5.3.2, we propose heuristics to reduce the complexity of scheduling general task graphs and extend Scheduling1D to a heuristic algorithm called Scheduling2D for scheduling general task graphs. Scheduling2D is again extended to deal with stochastic workloads in Section 5.3.3.

Table 6: The road map of our investigation

	Single Task	Task Graphs
Deterministic	STREAM-MP-D-STMS (Section 5.2.1)	STREAM-MP-D-TG Scheduling2D (Section 5.3.2)
Stochastic	STREAM-MP-S-STSMS (Section 5.2.2)	STREAM-MP-S-TG SScheduling2D (Section 5.3.3)

5.2 SCHEDULING A SINGLE TASK

There are times when a streaming application cannot be parallelized or simply does not provide a task-graph representation to a system. In this section, we consider scheduling

a single task on a multiprocessor system to attack the problems of STREAM-MP-D-ST and STREAM-MP-S-ST. The difference between these two problems is whether the task workload is deterministic or stochastic. Since there is only a single task, we will omit the subscripts of the parameters of the task unless confusion arises.

In scheduling a streaming application represented by a single task, there is no parallelism inside the application that we can exploit. This implies that the entire application has to be executed on one processor. Therefore, the maximum time to process a request in the worst case using a single processor at the maximum speed is less than the response time requirement (i.e., $\frac{W}{f_M} \leq D$). Otherwise, this application is not schedulable under the QoS requirements. Obviously, if the period is no less than the response time requirement (i.e., $T \geq D$), we need to use only one processor to execute this application and there is no point of using more than one processor. The problem of STREAM-MP-D-ST (STREAM-MP-S-ST) is essentially reduced to the problem of STREAM-UP-D-ST (STREAM-UP-S-ST).

It is more interesting to deal with the cases where the period is less than the response time requirement (i.e., $T < D$). If we still use only one processor, the application has to finish processing a request in time T. That is, the application is forced to process requests faster than the response time requirement, which results in increased dynamic energy consumption. Even worse, if $T < \frac{W}{f_M}$, the application is not schedulable under the QoS requirements. To resolve this problem, we can take advantage of availability of multiple processors. The resulting scheme is a master-slave scheme, the basic idea of which is to execute different instances of the task on multiple processors. Using more processors results in increased static energy consumption, but it will decrease the dynamic energy consumption. To optimize the total energy, a perfect balance must be struck between static and dynamic energy. Note that traditionally task duplication has been used as a technique to minimize the execution time of a task graph (e.g., [31]). However, we use it in this dissertation mainly as an energy-reduction approach.

In the master-slave scheme (Figure 17), we use a processor that acts as the master to receive requests and distribute them in a round-robin fashion to other processors, each acting as a slave and running a copy of the task. The master can be placed on the administrative processor in the system (e.g., the PPE in CELL [1]); Or, because of the light workload of

the master, in general, it can be placed with one slave on a processor. Therefore, we ignore the energy consumption of the master in the analysis. We assume that the time to send a request from the master to a slave is constant and we incorporate it into the response time requirement.

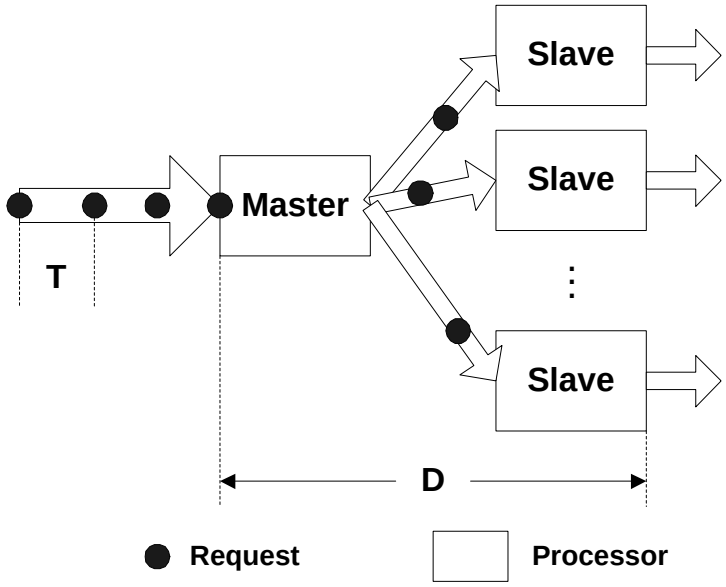
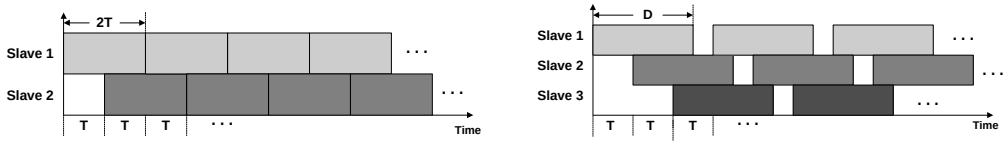


Figure 17: The master-slave Scheme

All active processors are symmetric in the sense that each processor employs the same speed schedule. Once the number of active processors is determined, the problem of STREAM-MP-D-ST is reduced to STREAM-UP-D-ST and the problem of STREAM-MP-S-ST is reduced to STREAM-UP-S-ST. To see why this is the case, let us look at a simplified example under the ideal processor model. Figure 18 shows different scenarios with different number of active processors for a streaming application whose response time requirement is 2.5 times the request interarrival time (i.e., $D = 2.5T$). Suppose that each request takes exact W cycles to process. If we use 2 slaves for this application (Figure 18(a)), the processing time for each request is $2T$ (note that $2T < D$) and the operating frequency of each processor is $\frac{W}{2T}$. If we use 3 slaves (Figure 18(b)), the processing time for each request is $min(D, 3T) = D$ (each processor will have $0.5T$ of idle time between consecutive requests) and the operating frequency of each processor is $\frac{W}{D}$.



(a) Using 2 slaves

(b) Using 3 slaves

Figure 18: Applying the master-slave scheme to a streaming application for which $D = 2.5T$

From the above example, we can see that the key question for the master-slave scheme is how to determine the optimal number of active processors to minimize the energy consumption of all the slaves. Suppose that the number of active processors is n, each active processor acts as a frame-based real-time system of frame length $min(nT, D)$. This also implies that the maximum number of active processors is $\lceil \frac{D}{T} \rceil$. If the number of active processors increases beyond $\lceil \frac{D}{T} \rceil$, the frame length will stay at D. As a result, the dynamic power consumption of each processor stays the same while the static power consumption increases. Thus, the number of active processors ranges from 1 to $\lceil \frac{D}{T} \rceil$. To obtain the optimal number of active processors, we also need to apply existing uniprocessor scheduling algorithms to compute the dynamic energy consumption. Next, we describe how to determine the optimal number of active processors under different scenarios.

5.2.1 Deterministic Workload

5.2.1.1 Ideal Processor Model Suppose that there are totally N requests for the streaming application. Let the number of active processors to service the requests be denoted by n, where $1 \leq n \leq \lceil \frac{D}{T} \rceil$.

The time allotted for servicing each request is $t = min(D, nT)$, and the time to service N requests is $(N-1) \cdot T + t$. Thus, the static energy consumption is

$$e_s(n) = n \cdot c_0 \cdot ((N-1) \cdot T + t) \approx n \cdot c_0 \cdot (N-1) \cdot T \approx nc_0NT$$

and the dynamic energy consumption is

$$e_d(n) = Nc_1 \left(\frac{W}{t}\right)^{\alpha} \times t$$

where $\frac{W}{t}$ is operating frequency of the processors. The total energy consumption is

$$e(n) = e_s(n) + e_d(n) = N\left(nc_0T + c_1\left(\frac{W}{t}\right)^{\alpha} t\right)$$

To obtain the optimal number of active processors n^*, we first relax two constraints. That is, we ignore the response time requirement and allow fractional number of processors. Thus the total energy consumption becomes

$$e(n) = N\left(nc_0T + c_1\left(\frac{W}{nT}\right)^{\alpha} nT\right)$$

Since nc_0T is an increasing function in n and $c_1\left(\frac{W}{nT}\right)^{\alpha} nT$ is a decreasing function in n, there is only one global minimum for $e(n)$. Through the first derivative of $e(n)$, we can obtain the optimal (fractional) value of n

$$\tilde{n} = \frac{W}{T}\sqrt[\alpha]{\frac{c_1}{c_0}(\alpha - 1)} \tag{5.1}$$

Based on the property of $e(n)$, we have the following rules to decide optimal actual number of active processor n^*.

1. If $\tilde{n} \leq 1$, then $n^* = 1$.
2. If $\tilde{n} \geq \lceil \frac{D}{T} \rceil$, then $n^* = \lceil \frac{D}{T} \rceil$.
3. Otherwise, the optimal number of active processors is either $\lfloor \tilde{n} \rfloor$ or $\lceil \tilde{n} \rceil$, and can be simply determined by comparing $e(\lfloor \tilde{n} \rfloor)$ and $e(\lceil \tilde{n} \rceil)$.

Thus, determining the optimal number of active processors under the ideal processor model can be done in constant time. All active processors will use the same speed to execute the application, which is $\frac{W}{min(D, n^*T)}$.

5.2.1.2 Realistic Processor Model The master-slave scheme under the realistic model is called the MS scheme, which is our solution to the problem of STREAM-MP-D-ST. To obtain the MS scheme, we first compute the total energy consumption $e(n)$ of processing N requests for each number of active processors n. The derivation of $e(n)$ is similar to that for the ideal processor model. The time allotted for servicing each request is $t = min(D, nT)$, The static energy consumption is

$$e_s(n) = n \cdot p_{idle} \cdot ((N-1) \cdot T + t) \approx n \cdot p_{idle} NT$$

With a slight abuse of notation, we use $\lceil s \rceil$ to denote the closest discrete speed higher than s. Thus, the dynamic energy consumption is

$$e_d(n) = N \cdot p\left(\lceil \frac{W}{t} \rceil\right) \frac{W}{\lceil \frac{W}{t} \rceil}$$

The optimal number of active processors n^* is

$$n^* = \underset{1 \le n \le \lceil \frac{D}{T} \rceil}{argmin} \left(e_s(n) + e_d(n)\right)$$

Unlike the case for the ideal model, we do not have a closed form solution to computing n^* for the realistic model. However, we can simply try all possible number of active processors ranging from 1 to $\lceil \frac{D}{T} \rceil$ to find n^*. Obviously, the time complexity of this approach is $O(\lceil \frac{D}{T} \rceil)$.

5.2.2 Stochastic Workload

Determining the optimal number of active processors for stochastic workload is very similar to that for deterministic workload. As in the case of deterministic workload, the number of active processors decides the static energy consumption and the deadline for processing each request. Once the deadline is known, we can apply the PACE scheme [40] for the ideal processor model, or the PPACE scheme described in Section 4.3.1 for the realistic processor model, to compute the dynamic energy for processing each request.

Thus, for the ideal processor model, we follow similar derivation as in Section 5.2.1 and apply Formula (4.4) to obtain the optimal (fractional) number of active processors

$$\tilde{n} = \frac{1}{T} \sqrt[\alpha]{\frac{c_1 \sum_{j=1}^{W} F_j^{\frac{1}{\alpha}}}{c_0}(\alpha - 1)}$$

We can follow the same rules in Section 5.2.1 to decide the optimal number of active processors n^*, which also takes constant time.

For the realistic processor model, we can use the same brute force approach as in Section 5.2.1. The resulting scheme is called the SMS (Stochastic MS) scheme, which is our solution to the problem of STREAM-MP-S-ST. In the SMS scheme, the PPACE scheme is applied in computing dynamic energy consumption. The time complexity of the SMS scheme is $O(\frac{D}{T}\frac{r^2 M \ln\lambda \ln(r\ln\lambda)}{\epsilon})$ according to the time complexity of PPACE in Section 4.3.1.

5.2.3 Evaluation

When scheduling a streaming application represented by a single task, making use of multiple processors serves two purposes. One is to satisfy the throughput requirement when the time to process a request using maximum speed is greater than the period. In this case, we are forced to use multiple processors. The other purpose is to reduce dynamic energy (while increasing static energy) to minimize the total energy consumption. We are more interested in the latter. In this section, we quantify the impact of using multiple processors in energy reduction in our proposed schemes through experiments, as follows.

Power models. For the processor power model in the experiments, we used Intel XScale [64]. The static power of the processor was varied to reflect the percentages of the static power in total power being 22%, 44%, and 67% for the 70nm, 50nm, and 35nm technologies, respectively [21].

Experiments on deterministic workload. We first compare the MS scheme and the uniprocessor scheme [7] for a single task. Note that if the MS scheme turns out to use only one processor, it is essentially equivalent to the uniprocessor scheme for a single task. The number of execution cycles for the task ranges from 1 million to 100 million. For the period T of the task, we chose 20 values distributed evenly between the minimum possible execution time (using the maximum speed) and half of the maximum possible execution time (using the minimum speed). For the deadline D of the task, we chose 20 values distributed evenly between twice the minimum possible execution time and the maximum possible execution time. Thus, we have $20 \times 20 = 400$ combinations of T and D. However, we only experiment with those combinations for which $2T \leq D$ since otherwise the MS scheme will use one processor and be equivalent to the uniprocessor scheme.

We find that the experimental results for different number of execution cycles are very similar. This can be mostly explained by Formula (5.1) and the way we chose the value of T. That is, the chosen values of T are all proportional to W. The optimal (fractional) numbers of processors are the same for different number of execution cycles. Thus, it is sufficient to show the results for the number of execution cycles being 1 million.

For 50nm and 35nm technologies, the MS scheme always uses only one processor because of the large static power. Thus, the MS scheme provides no savings over the uniprocessor scheme. For 70nm technology, the MS scheme could use up to 2 processors and provides up to 23.5% of energy saving over the uniprocessor scheme when the period is small (Figure 19). However, when the period becomes larger and so does the deadline, the MS scheme will use only 1 processor again. The experiment results verifies the fact that the smaller the static power, the more processors the MS scheme could end up using (also depending on the two QoS requirements).

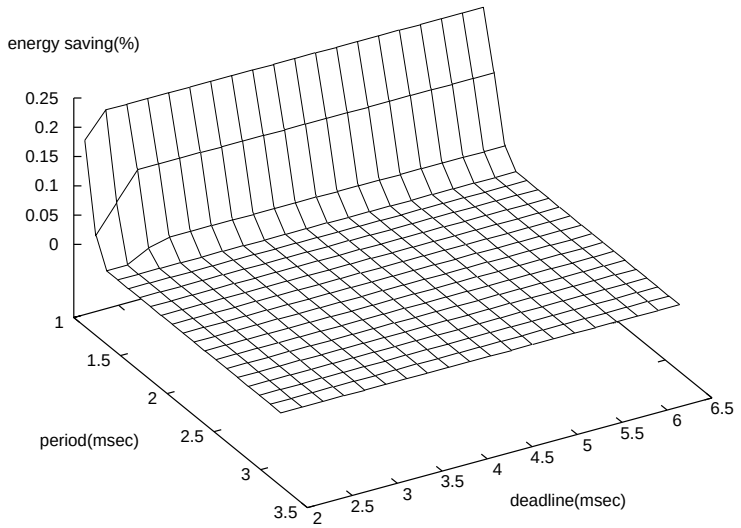


Figure 19: Energy savings for 70nm technology

Experiments on stochastic workload. We also compare the SMS scheme with uniprocessor scheme (PPACE) for a single task. We use the six distributions described in Section 3.6 for the task cycle distributions. The values of T and D are generated in the same way as in the case of deterministic workload.

We find that in all experiments, the SMS scheme uses only one processor and thus there are no savings over the uniprocessor scheme. The result is consistent with the previous result on deterministic workload. This is because if we compare two tasks that have the same number of worse-case execution cycles, but one has deterministic workload and the other has stochastic workload, the dynamic energy consumption of the former is greater than that of the latter when the deadlines are the same. Thus, the SMS will tend to use fewer processors than the MS scheme.

5.3 SCHEDULING A TASK GRAPH

The previous section assumes that a streaming application is represented by a single task, and thus there is no parallelism that we can exploit inside the application. In this section, we consider scheduling a streaming application represented by a task graph to attack the problems of STREAM-MP-D-TG and STREAM-MP-S-TG. The structure of a task graph exposes the parallelism in time (indicated by predecessor and successor relationship in the task graph) and the parallelism in space (indicated by sibling relationship in the task graph). We apply pipelining technique to exploit the parallelism in time and use parallel processing technique to exploit the parallelism in space, for the purpose of saving energy. We first consider task graphs with deterministic workload and propose a scheme called Scheduling2D to solve the problem of STREAM-MP-D-TG. We then extend the Scheduling2D scheme to deal with task graphs with stochastic workload to attack the problem of STREAM-MP-S-TG.

5.3.1 Scheduling for Linear Task Graphs with Deterministic Workload

In this section, we present the scheduling algorithm for a special type of task graphs, linear task graphs. We start with scheduling for linear task graphs because it serves as the basis for our scheduling algorithm for general task graphs.

In a linear task graph, task τ_1 is the source, τ_n is the sink, and the only predecessor of task τ_i $(1 < i \leq n)$ is task τ_{i-1} (that is, tasks can be arranged to form a straight line and a total order can be established on all tasks). Thus, only pipelining can be explored for linear task graphs.

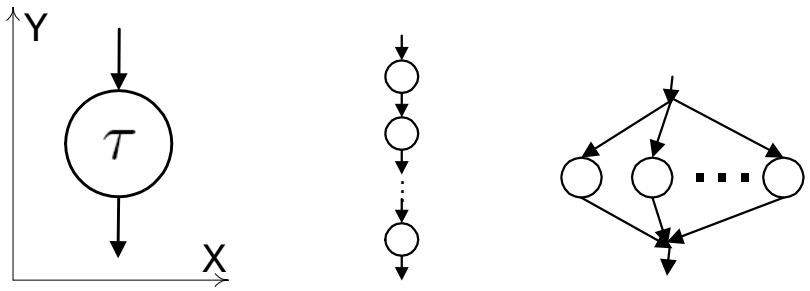
Although optimal scheduling of linear task graphs is NP-hard [5], it can be approximately solved by a fully polynomial time approximation scheme. That is, the solution returned by the scheduling algorithm is guaranteed to be within ϵ (ϵ is a user-defined parameter) of the optimal solution and the scheduling algorithm runs in time polynomial in $\frac{1}{\epsilon}$.

5.3.1.1 Y-Oriented Load To gain some insight, we simplify the problem of scheduling a linear task graph by relaxing the application model and applying the ideal processor model. We relax the application model by assuming that a streaming application is represented by a single task τ, following the *divisible load* model [11] (Figure 20(a)). In other words, although all W cycles of τ can only be executed consecutively, the task can be arbitrarily partitioned into any number of load fractions that have precedence relations. We call this Y-oriented load (Figure 20(b)) because it is depicted in the direction of Y-axis. There is also a notion of X-oriented load, which will be described in Section 5.3.2.1. Moreover, we ignore all communication cost.

Recall from Section 3.3.1 that the processor power function in the ideal model is

$$p(f) = c_0 + c_1 f^{\alpha} \tag{5.2}$$

where f is the operating frequency, constant c_0 represents the static power, and the term $c_1 f^{\alpha}$ represents the dynamic power.



(a) load (b) Y-oriented (c) X-oriented

Figure 20: Divisible load

To schedule τ, we first ignore the deadline constraint and consider the problem of energy minimization of the load τ subject to only the throughput requirement, $\frac{1}{T}$. Pipelining is a natural approach to satisfying the throughput requirement and load balancing is desired due to the convexity of the power function in Equation (5.2). Suppose that y processors are used to execute this load. Thus, each processor (corresponding to a pipeline stage) is

assigned $\frac{W}{y}$ cycles and required to execute these cycles in time T. Thus, the speed for each processor given only throughput requirement is $\frac{W/y}{T} = \frac{W}{yT}$. In servicing a single request, the static energy consumption is yc_0T and the dynamic energy consumption is $yc_1\left(\frac{W}{yT}\right)^{\alpha}T$. Therefore, the total energy consumption for servicing a request is

$$e_Y(y) \;=\; yc_0T + \frac{c_1W^{\alpha}}{y^{\alpha-1}T^{\alpha-1}} \tag{5.3}$$

which is a unimodal function and has a global minimum. This shows that starting from a single stage, deepening the pipeline will reduce the energy consumption while satisfying the throughput requirement $\frac{1}{T}$, until the number of pipeline stages increases past a certain value y^*, which is the optimal number of pipeline stages for load τ. The optimal number of pipeline stages strikes a balance between static and dynamic power. In fact, by obtaining the first derivative of $e_Y(y)$ and equating it to zero, we have the optimal number of pipeline stages for executing τ, which is given by

$$y^* = \sqrt[\alpha]{\frac{c_1(\alpha-1)}{c_0}} \cdot \frac{W}{T} \tag{5.4}$$

Note that for the purpose of describing the basic idea succinctly, we allowed ourselves not to be rigorous, that is, we can use fractional number of processors and do not consider the boundary conditions.

Suppose that we now impose the deadline constraint D on τ. If $D = T$, we have to use only 1 pipeline stage; if $D = 2T$, we can use 2 pipeline stages and the energy consumption is reduced. This shows that the difference between T and D can have impact on energy reduction. Ideally, to reduce the energy consumption of τ, $D > y^*T$. In this case, the response time of τ is y^*T, which is less than the deadline constraint D. This shows that the static power affects the upper bound of the response time when the goal is to save energy, and sometimes the application needs to service requests faster than the response time requirement to save energy. This is counter-intuitive because common wisdom on DVS scheduling says that the execution of tasks should be stretched as much as possible as long as the deadline constraint is not violated.

5.3.1.2 The Scheduling1D Algorithm We now revert back to the realistic processor model to design scheduling algorithm to schedule a linear task graph. Since linear task graphs are analogous to Y-oriented load and there is only parallelism in time, we call our scheduling algorithm for linear task graphs *Scheduling1D*.

The Structure of the Optimal Solution The basic scheduling strategy for linear task graphs is pipelining. There are three questions that need to be answered in order to schedule a linear task graph.

1. How many pipeline stages should be used to execute this task graph? Each pipelining stage will correspond to a processor. From the analysis in Section 5.3.1.1, we can see that the static power consumption has a significant impact on the optimal number of pipeline stages.
2. How to map the tasks in the task graph to processors? Obviously, only consecutive tasks in the task graph can be mapped to the same processor (i.e., a pipeline stage). Note that now the mapping granularity is tasks other than cycles as in Section 5.3.1.1 and that, due to the communication energy and delay, load balancing is not necessarily desired.
3. What speed is to be used for each processor such that the delay of each stage is no more than the period and the total delay of all stages is no more than the deadline? Note that due to the convexity of the power function and the fact that there is no communication cost among the tasks on the same processor (a pipeline stage), we make a simplified assumption that all tasks on the same processor will use the same speed.

The above three questions are correlated and should not be considered separately. Thus, the scheduling algorithm needs to perform finding the optimal number of stages, mapping, and speed scheduling simultaneously. We first present an optimal scheduling algorithm for linear task graphs. This optimal algorithm has worst-case exponential time complexity. Below, we propose an approximation algorithm that is based on the optimal algorithm.

The optimal scheduling algorithm for linear task graphs is based on the recursive structure of the optimal solution. Let the optimal scheduling of the tasks τ_i through τ_n when the end-to-end delay from task τ_i to task τ_n is t be denoted by the vector-valued function

$E_i(t) = [e, q, j, d]$, where e denotes the minimum energy consumption executing the tasks τ_i through τ_n when servicing a single request, q denotes the optimal number of stages for the tasks τ_i through τ_n, j indicates that tasks $\tau_i, \tau_{i+1}, \ldots, \tau_j$ are mapped to the first stage of the q stages, and d is the time used for the first stage plus the communication delay from the first stage to the next stage. With a slight abuse of notation, we use $E_i(t).e$ to denote the e component of $E_i(t)$ (similar notations can be obtained for other values of $E_i(t)$). Suppose that we are given the functions $E_{i+1}(\cdot)$ through $E_n(\cdot)$, we can compute $E_i(t)$ using the pseudo-code in Algorithm 5.1 (note that $v_{n,n+1} = 0$, that is, there is no communication after the sink).

Algorithm 5.1 Computing $E_i(t)$

$E_i(t).e := \infty$
{n is # of tasks}
for $j := i$ to n **do**
  $W := \sum_{k=i}^{j} W_k$
  {M is # of frequencies}
  for $s := 1$ to M **do**
    $d := \frac{W}{f_s} + t_p + \lambda v_{j,j+1}$
    if $d \leq T$ **then**
      {e_1 is the energy for the 1^{st} stage}
      $e_1 := (p(f_s) - p_{idle})\frac{W}{f_s} + p_{idle}T$
      $e := e_1 + \gamma v_{j,j+1} + E_{j+1}(t - d)$
      if $e < E_i(t).e$ **then**
        $E_i(t).e := e$
        $E_i(t).q := E_{j+1}(t - d).q + 1$
        $E_i(t).j := j$
        $E_i(t).d := d$
      end if
    end if
  end for
end for

The computation of the energy for the first stage at Line 8 in Algorithm 5.1 needs further clarification. The first term $(p(f_s) - p_{idle})\frac{W}{f_s}$ is the dynamic energy consumption of the first stage for servicing a single request. The second term $p_{idle}T$ is the static energy consumption, because the request lasts for the whole period T and p_{idle} is the power always consumed in a processor when it is on.

The optimal energy consumption and scheduling information of the whole linear task graph will be obtained from $E_1(D)$. We can see from Algorithm 5.1 that, in order to compute $E_1(D)$, we need to compute $E_2(\cdot), E_3(\cdot), \ldots, E_n(\cdot)$. In general, $E_i(\cdot)$ depends on $E_k(\cdot)$ $(k = i+1, \ldots, n)$. Thus, the base case is $E_n(\cdot)$, which denotes the scheduling information for a single task, τ_n. It is not difficult to see that the base case is a *step function* (piece-wise constant function) because there are only a limited set of discrete speeds available in processors. By induction, we can show that all functions $E_i(\cdot)$ are step functions. A step function can be represented by the end points of intervals in the function. Once all end points in a step function are identified, we can obtain any value of that step function. Because of the discrete nature and recursive structure of $E_i(\cdot)$, we can apply dynamic programming to compute these functions. We compute the functions $E_i(\cdot)$ $(i = 1, 2, \ldots, n)$ in reverse order. That is, we first compute $E_n(\cdot)$, then compute $E_{n-1}(\cdot)$, ..., and last compute $E_1(\cdot)$. Note that computing function $E_i(\cdot)$ is to identify all its end points, rather than compute a single value as in Algorithm 5.1.

The Optimal Scheduling Algorithm Before we present the algorithm to compute $E_i(\cdot)$, we refer readers to Section 4.4.3.2 for the representation of step functions and the associated operations. For succinct presentation, we do not show the computation of functions $E_i(\cdot).q$, $E_i(\cdot).j$ and $E_i(\cdot).d$ because they can be easily performed as a by-product of computing $E_i(\cdot).e$. We will also write $E_i(\cdot).e$ as $E_i(\cdot)$.

In Algorithm 5.1, we computed a single value of $E_i(\cdot)$. Now we make use of step functions to compute the whole function $E_i(\cdot)$ (i.e., identify all the end points in $E_i(\cdot)$). To do that, we first consider $(n-i+1) \times M$ helper functions $\hat{E}_{i,j,s}$ $(j = i, i+1, \ldots, n$ and $s = 1, 2, \ldots, M)$, where M is the number of available discrete speeds. $\hat{E}_{i,j,s}$ denotes the energy function when tasks $\tau_i, \tau_{i+1}, \ldots, \tau_j$ are mapped to the first stage and f_s is the speed used in the first stage.

A single value of $\hat{E}_{i,j,s}$ can be computed as

$$\begin{aligned}\hat{E}_{i,j,s}(t) &= (p(f_s) - p_{idle})\frac{\sum_{k=i}^{j} W_k}{f_s} + p_{idle}T + \gamma v_{j,j+1} \\ &\quad + E_{j+1}(t - \frac{\sum_{k=i}^{j} W_k}{f_s} - t_p - \lambda v_{j,j+1})\end{aligned} \tag{5.5}$$

Computing the whole function $\hat{E}_{i,j,s}$ can be expressed using step functions described in Section 4.4.3.2 as Line 9 in Algorithm 5.2. The desired function E_i is obtained by merging the $(n-i+1) \times M$ $\hat{E}_{i,j,s}$ functions. During the merging process, the optimal values of $E_i(\cdot).q$, $E_i(\cdot).j$ and $E_i(\cdot).d$ corresponding to each point are also determined. The optimal scheduling algorithm for linear task graphs is shown at Lines 1-12 in Algorithm 5.2. Line 16 is used for the approximation algorithm.

We now analyze the time complexity and space complexity of computing E_i. Computing the helper function $\hat{E}_{i,j,s}$ takes time $O(|E_{j+1}|)$ and the number of points in $\hat{E}_{i,j,s}$ is also $O(|E_{j+1}|)$. The key operation in computing E_i is the merge operation over $(n-i+1) \times M$ step functions, each is of size $O(|E_{i+1}|)$. Thus, the time to compute E_i is $O(nM|E_{i+1}| \log^2 nM)$ and the number of points in E_i is $O(nM|E_{i+1}|)$. Since the base case is $|E_{n+1}| = 1$, we can obtain the closed forms of the time complexity and space complexity of computing E_i to be $O((nM)^{n-i+1} \log^2 nM)$ and $O((nM)^{n-i+1})$, respectively. Since the optimal solution is in $E_1(\cdot)$, the time complexity and space complexity of the optimal scheduling algorithm for linear tasks graphs are $O((nM)^n \log^2 nM)$ and $O((nM)^n)$, respectively.

Approximation Algorithm The time complexity of the optimal scheduling algorithm for linear task graphs depends greatly on the size of functions E_i. As we can see from the analysis of the optimal scheduling algorithm, the size of E_i may grow exponentially as i goes from n to 1. Thus, we need to control the size of function E_i within some polynomial bound. To do that, we apply similar approach as in Section 4.4.3.2. That is, we trim function E_i using the TRIM procedure in Algorithm 4.3 with the value of the parameter $\delta = (1+\epsilon)^{\frac{1}{n}} - 1$. By using the same reasoning and derivation, we can reach that $|E_i| = O(\frac{n \log \lambda}{\epsilon})$, where $\lambda = \frac{\mathbb{P}_l.e}{\mathbb{P}_r.e}$. Thus, the number of points in E_i is upper bounded by a polynomial in $\frac{1}{\epsilon}$.

Algorithm 5.2 Scheduling1D(ϵ)

1: $E_{n+1} := \{(0,0)\}$
2: **for** $i := n$ downto 1 **do**
3: {compute E_i}
4: **for** $j := i$ to n **do**
5: $c := \sum_{k=i}^{j} c_k$
6: **for** $s := 1$ to M **do**
7: $d := \frac{c}{f_s} + t_p + \lambda v_{j,j+1}$
8: **if** $d \leq T$ **then**
9: $\hat{E}_{i,j,s} := (P(f_s) - P_{idle})\frac{c}{f_s} + P_{idle}T + \gamma v_{j,j+1} +_e (d +_t E_{j+1})$
10: **else**
11: $\hat{E}_{i,j,s} := \phi$
12: **end if**
13: **end for**
14: **end for**
15: $E_i := \bigcup_{j=i,\ldots,n,s=1,\ldots,M} \hat{E}_{i,j,s}$
16: $E_i := TRIM(E_i, (1+\epsilon)^{\frac{1}{n}} - 1)$
17: **end for**

5.3.2 Scheduling for General Task Graphs With Deterministic Workload

In this section, we present the scheduling algorithm for general task graphs. For general task graphs, there exists not only temporal parallelism (Y-oriented load), but also spacial parallelism (which we call X-oriented load). Thus, we first study X-oriented load by relaxing the application model and applying the ideal processor model, as in Section 5.3.1.1, to gain insight into the problem. Then we provide two heuristics to reduce the complexity of scheduling general task graphs. Finally, we present the complete scheduling algorithm.

5.3.2.1 X-Oriented Load We relax the application and power models in the same way as in Section 5.3.1.1 except for assuming that all W cycles of τ can be executed in parallel (*X-oriented load* (Figure 20(c)). Now we look at the problem of energy minimization of the load τ subject to the deadline constraint, D. Parallel processing is a natural approach to satisfying the deadline constraint and load balancing is desired due to the convexity of the power function. Suppose that x ($x \leq \mathbb{N}$) processors are used to execute this load. Thus, each processor is assigned $\frac{W}{x}$ cycles and its corresponding speed is $\frac{W/x}{D} = \frac{W}{xD}$. The static energy consumption is xc_0D and the dynamic energy consumption is $xc_1\left(\frac{W}{xD}\right)^3 D$. Therefore, the total energy consumption is $e_X(x) = xc_0D + \frac{c_1W^3}{x^2D^2}$, which is very similar to Equation (5.3). The optimal number of active processors for executing τ is

$$x^* = \sqrt[3]{\frac{2c_1}{c_0}} \cdot \frac{W}{D} \tag{5.6}$$

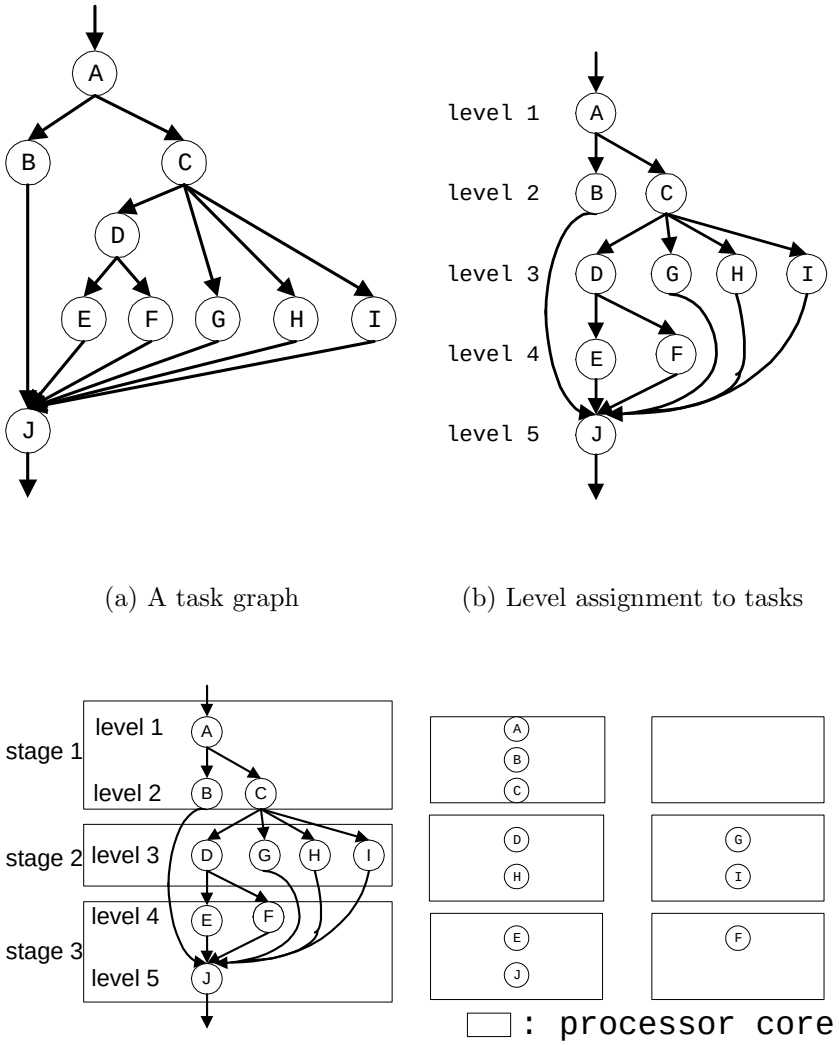
Analogously to Y-oriented load, starting from uniprocessor, increasing the degree of parallel processing can initially reduce the energy consumption by reducing the dynamic energy of the processors while satisfying the deadline constraint D. However, the energy consumption will start to increase after the degree of parallelism increases past a certain value due to the static energy used by too many processors. The optimal degree of parallelism strikes a balance between static and dynamic power.

5.3.2.2 Scheduling Heuristics A general task graph can be roughly regarded as a mixture of X-oriented load and Y-oriented load. For X-oriented load, we use parallel processing to reduce energy consumption while satisfying the deadline constraint; for Y-oriented load, we use pipelining to reduce energy consumption while satisfying the throughput requirement. Thus, the first step of our scheduling algorithm is to identify the X-oriented load and Y-oriented load of the task graph. To this end, our *first heuristic* is to use the classical topological sort to assign a level to each node in the task graph. The level of a node (task) is equal to 1 plus the length of the longest[1] path from the source to this node. By assigning a level to each node in the task graph, we essentially morph the task graph into a two-dimensional structure. Figures 21(a) and 21(b) show an example of task graph morphing. The tasks on the same level represent the X-oriented load and the tasks across different levels represent the Y-oriented load.

To match the two-dimensional structure of the morphed task graph, we conceptually consider the processors to form a two-dimensional logical structure (Figure 21(d)). In mapping the task graph onto the processors, the X-dimension is used to map the X-oriented load and Y-dimension is used to map the Y-oriented load, while considering energy consumption in both dimensions. The logical arrangement of processors makes the underlying logical tiled structure the same as the structure of the task graph, which will make the mapping process computationally tractable. Our *second heuristic* is to let a row of processors in the logical tiled structure correspond to a pipeline stage in the mapping process, and we only allow contiguous levels of the morphed task graph to be mapped to the same pipeline stage. Figure 21(c) shows a possible mapping from levels to pipeline stages, and Figure 21(d) shows possible mapping from tasks to processors on each pipeline stage.

5.3.2.3 The Scheduling2D Algorithm Unlike previous work on energy-aware task graph scheduling, which separated task mapping and speed scheduling, we interweave these two closely correlated components. There are two types of mappings, each corresponding to a dimension. The first type of mapping is called *Y-mapping*, which is performed along the Y-dimension (pipelining dimension). Performing Y-mapping includes: (i) determining the

[1] We consider the longest path due to the precedence constraints.



(a) A task graph

(b) Level assignment to tasks

(c) Pipeline stage formation

(d) Final task mapping

Figure 21: An example of scheduling for general task graphs

optimal number of pipeline stages; (ii) allotting time to each stage while guaranteeing that the allotted time for each stage is no greater than T and that the sum of the allotted times for all stages is no greater than D; (iii) mapping levels to pipeline stages. The second type of mapping is called *X-mapping*, which is performed along the X-dimension (parallel processing dimension) for each stage. Performing X-mapping for each stage includes: (i) determining the optimal number of active processors for the stage; (ii) mapping tasks to active processors; (iii) deciding execution speed for each task while guaranteeing that all tasks finish executing and transferring data to their successors within the allotted time for the stage. Note that task speed scheduling occurs during X-mapping. Because of the mapping along two dimensions, we call the scheduling algorithm *Scheduling2D*.

We first explain X-mapping since it is less involved. X-mapping is mostly the classical multiprocessor scheduling problem (which is NP-hard) because, if the number of active processors for a stage is known, we can apply the classical list scheduling algorithm to approximate the load-balancing mapping, and then apply the techniques[2] in [24, 45, 5] to obtain the execution speed for each task. To determine the optimal number of active processors in a pipeline stage, it is straightforward to use a brute-force approach to check every possible number of active processors and find out the number of processors resulting in the minimum energy consumption. However, this approach is not suitable for large number of tasks. Instead, we apply hill-climbing methods to search for the best solution using Formula (5.6) as the starting estimation on the optimal number of active processors. This approach has much lower time complexity than the brute-force approach and our experiments show that the solutions obtained by this approach are very close to those obtained by the brute-force approach.

Y-mapping is very similar to scheduling linear task graphs if we treat each level in general task graphs as a task in linear task graphs. However, Y-mapping cannot be performed alone because it requires knowledge from performing X-mapping. Next, we will describe the details of Y-mapping.

Suppose that the total number of levels is L. Let the vector-valued function $E_i(t) = [e, q, j, d]$ ($i \leq j \leq L$ and $0 < d \leq t$) denote the scheduling of the tasks on level i to level

[2]All these techniques take into consideration communication among tasks

L given an allotted time t (end-to-end delay from level i to level L). In this scheduling, e denotes the energy consumption of the tasks on level i to level L, q denotes the number of stages needed for the tasks on level i to level L, j indicates that level $i, i+1, \ldots, j$ are mapped to the first stage of the q stages, and d is the time allotted to that stage (including the delay resulting from communication between that stage and the next stage), while level $j+1$ to level L are mapped to subsequent $q-1$ stages and the mapping information can be recursively obtained from $E_{j+1}(t-d)$. We can see that the definition of $E_i(\cdot)$ is very similar to that in Section 5.3.1.2 if we associate levels for general task graphs with tasks for linear task graphs. The scheduling algorithm, Scheduling2D, for general task graphs is shown in Figure 5.3. Scheduling2D can be regarded as an extension to Scheduling1D (Figure 5.2). The main difference between these two algorithms is the scheduling for the first stage of the q stages.

Algorithm 5.3 Scheduling2D(ϵ)

use topological sort to assign a level to each task
{L is the total number of levels}
$E_{L+1} := \{(0,0)\}$
for $i := L$ downto 1 **do**
  {compute E_i}
  for $j := i$ to L **do**
    compute W as the sum of cycles of the tasks on level i through level j
    compute m as the maximum degree of parallelism on level i through level j
    $I := \frac{W}{mf_M}$
    for $k := 1$ to $\lfloor \frac{T}{I} \rfloor$ **do**
      $d := kI$
      $\hat{E}_{i,j,k} :=$XMAP$(i,j,d) +_e (d +_t E_{j+1})$
    end for
  end for
  $E_i := \bigcup_{j=i,\ldots,L,k=1,\ldots,\lfloor T/I \rfloor} \hat{E}_{i,j,k}$
  $E_i := TRIM(E_i, (1+\epsilon)^{\frac{1}{L}} - 1)$
end for

Algorithm 5.4 XMAP(i, j, d)

1: use formula (5.6) to estimate number of processors
2: use an algorithm from [24, 45, 5] to perform mapping and speed scheduling for the tasks on level i to level j subject to real-time constraint d
3: use hill-climbing to search for a better solution
4: **return** the minimum energy consumption

For linear task graphs, because a stage corresponds to a single processor and all tasks in the same stage have the same speed, the time allotted to the first stage only has M possibilities, each corresponding to one of the available M discrete speeds. However, for general task graphs, the scheduling for the first stage is a case of X-mapping in which multiple processors may be used and different tasks may have different speeds. Enumerating every possible number of processors and possible speed for each task would result in exponential number of schedules for the first stage. Note that not all of such schedules are useful because if a schedule consumes more energy and uses more time than another schedule, then the former is useless.

Our approach is to use the alloted time for the first stage directly to decide the schedule. For any given allotted time, X-mapping will attempt to find the schedule with minimum energy consumption. We need to discretize the allotted time to make the scheduling for the first stage tractable. We use a heuristic to choose the discretization interval. For a given stage, let W be the sum of the cycles of all tasks in this stage and m be the maximum degree of parallelism for this stage (equal to the maximum number of tasks on any level that is mapped to this stage). We use the discretization interval of $\frac{W}{mf_M}$, which can be regarded as the minimum possible allotted time for this stage.

5.3.3 Scheduling General Task Graphs with Stochastic Workload

In this section, we consider the problem of STREAM-MP-S-TG, that is, scheduling general task graphs with stochastic workload. In the presence of stochastic workload, the objective becomes minimizing expected energy consumption. Dynamic slack reclamation has been

shown to be indispensable in dealing with stochastic workload. For uniprocessor systems, dynamic slack is reclaimed across tasks in the same processor. In the case of multiprocessor systems, dynamic slack is also reclaimed across processors. Our approach solution to STREAM-MP-S-TG is an extension to the Scheduling2D algorithm, which we call the SScheduling2D (Stochastic Scheduling2D) algorithm. Next, we describe the offline part and the online part of SScheduling2D in succession.

5.3.3.1 The Offline Part of SScheduling2D

As in the case of uniprocessor systems, one is tempted to extend the offline part of Scheduling2D by incorporating dynamic slack reclamation and comparing expected energy among different mappings and speed schedules. More specifically, the function $XMAP$ in the Scheduling2D algorithm would compute the expected energy consumption of a particular stage and the function E_i would denote the optimal expected energy consumption of the tasks from level i to level L. An important assumption of Scheduling2D is that E_i is agnostic of tasks on levels before level i and its value is only dependent on the time allotted to level i to level L. Unfortunately, dynamic slack reclamation across processors makes this assumption invalid. This is best demonstrated by the following example.

Suppose that we have a linear task graph that consists of only two tasks, τ_1 and τ_2. The communication cost is ignored. The requests come in every 2 time units (i.e., $T = 2$) and the response time requirement is 4 time units (i.e., $D = 4$). Two stages (i.e., two processors) are used to execute this task graph. Task τ_1 is executed on Processor 1 and could execute for 1 and 2 time units. Task τ_2 is executed on Processor 2 and has a constant workload. A possible execution scenario (Figure 22) is as follows.

1. At time 0, the first request arrives. Task τ_1 starts to execute on Processor 1.
2. At time 1, Task τ_1 finishes and produces 1 time unit of slack, which is reclaimed by Processor 2 for Task τ_2. Task τ_2 starts to execute on Processor 2 using a speed that makes it finish in 3 time units.
3. At time 2, the second request arrives, Task τ_1 starts to execute on Processor 1.
4. At time 3, Task τ_1 finishes and again produces 1 time unit of slack. However, Task τ_2 is still processing the first request and Processor 2 cannot reclaim this slack.

5. At time 4, Task τ_2 finishes processing the first request and starts to process the second request in 2 time units.

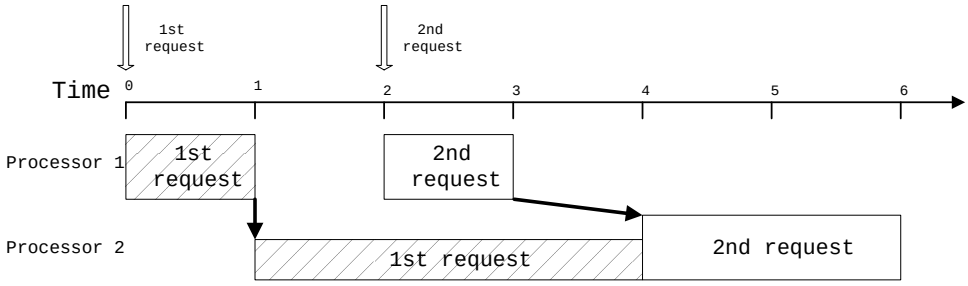


Figure 22: An execution scenario

In the above scenario, it takes the same amount of time (1 time unit) for Task τ_1 to process both requests, leaving the same amount of time (3 time units) for Task τ_2 to process both requests. However, Task τ_2 uses different energy for these two requests because it uses 3 time units to process the first request and only 2 time units to process the second.

Another difficulty of the aforementioned extension is computing the expected energy consumption in function *XMAP*. Because *XMAP* is in the inner loop of Scheduling2D, it requires low time complexity. However, other than enumerating all possible combination of speeds for all the tasks in a particular stage, we do not know of any good and low-complexity algorithm to accomplish this task.

Based on the above analysis, we can see that it is difficult and not clear how to incorporate dynamic slack reclamation into the offline part of Scheduling2D to compute the expected energy consumption. Thus, we choose to keep the offline part of Scheduling2D intact in SScheduling2D except for two simple extensions.

1. We use average number of execution cycles instead of worst number of execution cycles in the mapping phase of *XMAP*. The idea is to do the load balancing in a stage based on the average load of tasks in the hope to minimize the average (expected) energy consumption.
2. We store the start time (relative to the arrival time of a request) for each task for the online part of SScheduling2D to compute dynamic slack. Note that the start time of

each task is also the latest start time because we still use worst number of execute cycles to do the speed scheduling.

These two simple extensions do not add any time complexity and we can still guarantee that the application will satisfy the throughput and response time requirement in the worst case.

5.3.3.2 The Online Part of SScheduling2D In the online part of SScheduling2D, we employ dynamic slack reclamation and distribute the slack in a greedy fashion, that is, we give all the slack to the tasks available to execute. More specifically, when a task is ready to execute, we take the difference between the current time and the latest start time computed in the offline part as the dynamic slack given to this task. We apply the best intra-task DVS scheme, PPACE, to execute a task.

5.3.4 Experimental Results

In this section, we evaluate our proposed scheduling algorithms, Scheduling2D and SScheduling2D, through simulations. We first compare Scheduling2D against the previously existing algorithms for the problems similar to STREAM-MP-D-TG since no other work has proposed a solution to STREAM-MP-D-TG. We then compare SScheduling2D against Scheduling2D under stochastic workload. Multiple task graphs and different power models were employed in the evaluation, as follows.

Power models: For the processor model in the experiments, we used Intel XScale [64]. As in the evaluation of the MS and SMS schemes, the static power of the processor was varied to reflect the percentages of the static power in total power being 22%, 44%, and 67% for the 70nm, 50nm, and 35nm technologies, respectively [21]. Different values of static power result in different processor power model. For communication cost, we used a transmission rate of 20 Gbytes/s and the transmission power is set to 20% of maximum processor power when the communication link is fully utilized [59].

Evaluation of Scheduling2D Both synthetic and real-world task graphs were used in

the experiments. The synthetic task graphs are from TGFF and the real-world task graph is ATR, as described in Section 3.6. The X-mapping in our Scheduling2D algorithm is based on the S-SPM algorithm [45] because of its low time complexity and reasonably good performance. Also, we set the parameter ϵ to 0.05. We chose the latest work [5] on a subproblem of STREAM-MP-D-TG, which assumes that the period is equal to the deadline, to be the baseline against which Scheduling2D compared. A convex programming based approach[3] was used in [5] to obtain the execution speed for each task given the task mapping. Since it does not consider turning processor on/off, we enhanced it by trying all possible number of processors to find the minimum energy consumption. We used the classical earliest task first (ETF) list scheduling heuristic [70] to perform the task mapping.

We compare the two algorithms, baseline as described above and Scheduling2D, for different power models, different deadline constraints, and different throughput requirements. The values of period T and deadline D are generated similarly as in the case of evaluating the MS and SMS schemes. For the period T of synthetic task graphs, we chose 20 values distributed evenly between the shortest possible execution time (time to execute the critical path of the task graph using the maximum speed) and half of the time to execute the critical path of the task graph using the minimum speed. For the deadline D of the task graphs, we chose 20 values distributed evenly between twice the minimum possible execution time and the maximum possible execution time. Thus, we have $20 \times 20 = 400$ combinations of T and D. However, we only experimented with those combinations for which $2T \leq D$ since otherwise Scheduling2D is the same as the baseline. For ATR, the QoS requirements in its documentation are 2-33ms for the period and 2-1000ms for the deadline. In our experiments, we used values guided by this range and the processor model used: the period range is approximately 3ms-9ms and the deadline goes up to 18ms. Note that the baseline algorithm takes T as its deadline constraint because $T < D$.

[3] In some of the experiments, the convex program solver (called *ipopt*) that we used could not produce a solution due to its iteration limit. When that happened, we simply used S-SPM [45] in its place.

Table 7: Energy savings(%) of Scheduling2D over baseline

Task graph	70 nm				50 nm				35 nm			
	#CPUs	#stages	avg.	max	#CPUs	#stages	avg.	max	#CPUs	#stages	avg.	max
kseries_parallel	2 - 8	1 - 5	18.44	46.62	1 - 5	1 - 3	9.05	43.47	1 - 5	1 - 3	6.42	49.74
creds1	1 - 4	1 - 4	23.12	53.23	1 - 3	1 - 3	14.06	55.16	1 - 3	1 - 3	12.99	60.16
simple	1 - 5	1 - 4	16.26	41.34	1 - 3	1 - 3	6.93	32.47	1 - 3	1 - 3	4.62	32.9
kbasic_tables	1 - 6	1 - 4	17.89	37.86	1 - 4	1 - 3	10.42	26.19	1 - 3	1 - 3	7.56	25.61
kseries_parallel_xover	2 - 6	1 - 4	18.5	46.54	1 - 4	1 - 3	9.38	39.94	1 - 3	1 - 3	7.39	39.97
bugtest	3 - 11	1 - 4	18.87	38.22	2 - 8	1 - 3	9.1	23.36	2 - 6	1 - 3	6.74	27.79
kbasic_task	2 - 10	1 - 5	20.29	43.22	1 - 7	1 - 5	11.36	37.25	1 - 6	1 - 4	9.37	38.62
kextended	1 - 10	1 - 5	17.74	35.32	1 - 6	1 - 3	8.91	27.44	1 - 5	1 - 3	6.86	29.3
packets	1 - 4	1 - 2	18.6	40.38	1 - 4	1 - 2	8.55	30.65	1 - 3	1 - 2	6.1	30.98
ATR	2 - 7	1 - 3	14.98	35.66	1 - 4	1 - 2	6.28	17.95	1 - 3	1 - 3	3.42	9.7

Table 7 shows the energy savings of Scheduling2D over the baseline for all experiments. Scheduling2D achieves up to 53.2%, 55.1%, 60% savings for 70nm, 50nm, 35nm technologies, respectively. Even for average, Scheduling2D saves 18.5%, 9.4%, 7.1% for 70nm, 50nm, 35nm technologies, respectively. In general, it can be observed that as the static power increases, the energy saving obtained by Scheduling2D decreases. This is because high static power will force both algorithms to use fewer number of processors (for Scheduling2D, this translates to fewer number of pipeline stages), and thus the room for optimization is reduced.

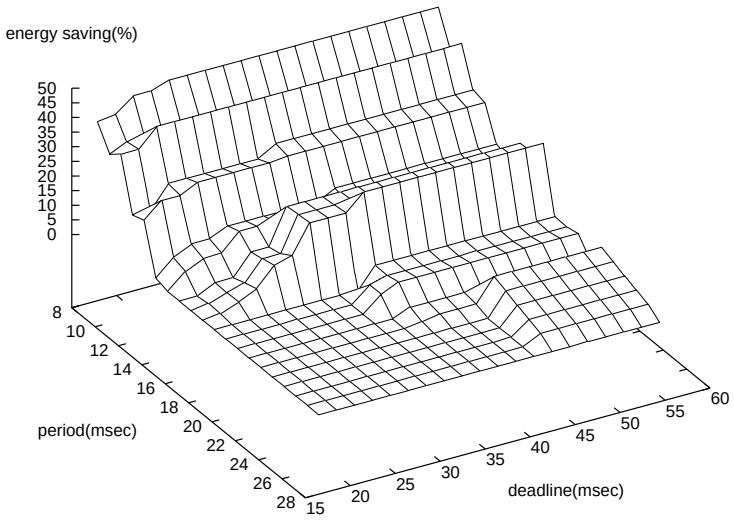


Figure 23: Energy savings for 70nm technology on k_series_parallel_xover

We show the effect of throughput and response time requirements on the *energy savings* of Scheduling2D over the baseline through task graph `k_series_parallel_xover` from TGFF (the results for other task graphs are similar). From Figure 23, we can see that as the period increases, the energy saving tends to decrease (however, we can spot some increase during the course due to the nature of discrete speeds). Increasing period means smaller workload per time unit. Thus, both algorithms tend to use lower speed and there is less room for optimization. When the period increases to the point where pipelining is not needed (i.e., Scheduling2D will only use one stage), Scheduling2D will essentially act as the baseline and the energy saving is reduced to zero. We can also see that for a given period, increasing

deadline will initially result in increased energy savings. This is because increasing deadline will allow Scheduling2D to use more pipeline stages to lower the energy consumption of the streaming application. However, as stated in Section 5.3.1.1, static power affects the upper bound of the response time and thus the upper bound of the number of pipeline stages. Therefore, we can observe from Figure 23 that continuing increasing deadline after certain point will not increase the energy saving since the number of pipeline stages will stay unchanged.

Evaluation of SScheduling2D We evaluated SScheduling2D through comparison against Scheduling2D on stochastic workload. Note that Scheduling2D is designed for deterministic workload. Thus, when it is applied to stochastic workload, the speed of each task is based on the assumption that each task runs for the worst-case number of execution cycles. For the stochastic workload used in the experiments, we only used synthetic task graphs in the experiments since the real-world task graph ATR has only deterministic workload. The task cycle distribution is the uniform distribution described in Section 3.5.

Table 8 shows the energy savings of SScheduling2D over Scheduling2D for all experiments. SScheduling2D achieves up to 36.67%, 37.64%, 19.17% savings for 70nm, 50nm, 35nm technologies, respectively. On average, SScheduling2D saves 9.03%, 12.38%, 6.75% for 70nm, 50nm, 35nm technologies, respectively. An interesting result is that energy savings do not necessarily decrease as the static power increases. This is because these two algorithms use the same number of processors in most test cases and thus result in the same static energy consumption. The main source of the energy savings of SScheduling2D is dynamic energy consumption.

Table 8: Energy savings (%) of SScheduling2D over Scheduling2D

Task graph	70 nm				50 nm				35 nm			
	#CPUs	#stages	avg.	max	#CPUs	#stages	avg.	max	#CPUs	#stages	avg.	max
kseries_parallel	2 - 8	1 - 4	9.3	24.47	1 - 5	1 - 4	12.15	30.2	1 - 4	1 - 3	6.97	15.41
creds1	1 - 6	1 - 4	8.95	36.67	1 - 3	1 - 3	12.34	37.64	1 - 3	1 - 3	5.74	17.84
simple	1 - 5	1 - 4	8.19	36.23	1 - 3	1 - 3	11.43	30.63	1 - 3	1 - 3	5.86	19.17
kbasic_tables	2 - 6	1 - 4	9.26	27.71	1 - 4	1 - 3	13.11	27.98	1 - 3	1 - 3	6.86	13.81
kseries_parallel_xover	2 - 6	1 - 4	8.65	24.29	1 - 4	1 - 3	12.99	28.19	1 - 3	1 - 3	6.76	15.8
bugtest	4 - 11	2 - 4	9.95	16.95	2 - 9	1 - 4	13.19	23.18	2 - 6	1 - 3	8.78	14.94
kbasic_task	2 - 12	1 - 5	8.95	34.29	1 - 7	1 - 4	12.57	29.28	1 - 6	1 - 4	7.06	16.75
kextended	2 - 9	1 - 4	8.32	24.44	1 - 6	1 - 3	12.02	26.77	1 - 4	1 - 3	6.94	18.34
packets	1 - 6	1 - 2	9.75	33.83	1 - 4	1 - 2	11.67	34.13	1 - 4	1 - 2	5.8	17.58

5.4 SUMMARY

In this chapter, we investigated energy-aware multiprocessor scheduling problems for streaming applications. In multiprocessor scheduling, we need to consider two issues that do not exist in uniprocessor scheduling. The first issue is how to find a balance between static and dynamic energy consumption because of the availability of multiple processors. The second issue is how to exploit the difference between the two QoS requirements, namely, throughput and response time. If we collapsed the two QoS requirements into one, as in the case for uniprocessor scheduling, we would lose the opportunity for energy optimization. In fact, addressing these two issues is the key to energy-aware multiprocessor scheduling problems.

We started out by investigating scheduling a streaming application represented by a single task. Although there is no parallelism that can be exploited inside the application, we proposed a master-slave scheme that executes different instances of the streaming application on different processors to satisfy the two QoS requirements while attempting to minimizing the total energy consumption. The key to the master-slave scheme is how to find the optimal number of active processors to execute the streaming application. We derived the formula and algorithm under different workloads and different processor models.

We then turned to scheduling a streaming application represented by a task graph. For deterministic workload, we proposed an algorithm called Scheduling2D that exploits the difference of the two QoS requirements to perform processor allocation, task mapping, and task speed scheduling simultaneously. Scheduling2D uses parallel processing and pipelining in the task mapping, as traditional algorithms for maximizing throughput or minimizing latency do. However, Scheduling2D focuses on finding the appropriate number of processors and allotting the optimal amount of time to each pipeline stage and each task in order to save energy. For stochastic workload, we extended Scheduling2D and proposed an algorithm called SScheduling2D that makes use of dynamic slack reclamation technique to minimize the expected energy consumption.

6.0 CONCLUSIONS

While traditionally performance has been the major concern for streaming applications, we are now witnessing a focus shift from pursuing maximum performance to energy-performance trade-off. This is because for battery-powered systems, increased energy consumption shortens operation time, and for high-end servers, increased energy consumption generates excessive amount of heat and reduces system reliability. Since streaming applications usually operate for long periods of time, energy optimization is especially important. Even a small improvement in scheduling algorithms can translate into significant energy savings.

This dissertation addresses the problem of scheduling a stream application with the goal of minimizing its energy consumption while satisfying two typical QoS requirements, namely, throughput and response time. An important feature of this dissertation is a complete treatment of the problem by taking into account different underlying platforms, different characteristics of workload, and different types of task graphs. Furthermore, an ideal and a realistic processor power models are considered. Although the ultimate goal is to obtain scheduling algorithms under the realistic model, considering the ideal model has proven to be very helpful (e.g., in the derivation of the PITDVS2 scheme). The easy mathematical manipulation of the ideal model often leads to elegant and optimal scheduling algorithms, which give great insight into the problem and provide the basis for designing practical scheduling algorithms.

In energy-aware scheduling of streaming applications on uniprocessor systems, the two QoS requirements essentially collapse into one and the problem is very closely related to energy-aware scheduling for frame-based hard real-time systems. One of the contributions of this dissertation to the state of the art in energy aware real-time scheduling theory is to show that simple patches to optimal scheduling algorithms obtained under simple models (e.g., the ideal model) do not necessarily generate scheduling algorithms that perform well

in practice. This is demonstrated multiple times throughout Chapter 4.

1. For frame-based systems with a single task, previously existing DVS schemes, such as PACE and GRACE, patch the solution obtained under the ideal model to comply with the realistic model. I propose a new DVS scheme called PPACE that is based directly on the realistic model. PPACE can give performance guarantees and achieve energy savings very close to the optimal solution. Experimental results show that PPACE outperform the existing schemes significantly.

2. For frame-based systems with multiple tasks, the PGOPDVS scheme, which is based on the optimal hybrid DVS scheme under the ideal model, achieves worse performance than all other DVS schemes under consideration in most of the experiments. Another example is the PITDVS scheme, which is based on the optimal inter-task DVS scheme under the ideal model. PITDVS is not necessarily better the DVS schemes that do not use probabilistic information of the workload (e.g., the Greedy scheme). However, the PITDVS2 scheme, which is based on PITDVS and uses two speeds to emulate a continuous speed, is shown to outperform the existing DVS schemes in the experiments.

The moral story is that any DVS scheme obtained through patching the solution under the ideal model needs to be justified by comparing against the optimal schemes, which can be obtained using our proposed unified approach. The unified approach is the culmination of the investigation on uniprocessor scheduling. It derives optimal (or provably close to optimal) stochastic inter-task, intra-task, and hybrid DVS schemes under the realistic model.

In energy-aware scheduling of streaming applications on multiprocessor systems, the two QoS requirements need to be distinguished in order to be satisfied as well as to save more energy. For scheduling a singleton task graph on multiprocessor systems, I propose a master-slave scheme that executes different instances of the streaming application on different processors. The purpose of using multiple processors in the master-slave scheme is twofold. One is to satisfy the throughput requirement and the other is to reduce energy consumption. Our experimental results show that beyond 50 nm technology, using multiple processors will not save energy comparing using a single processor because of the high static power. This implies that a streaming application needs to expose its parallelism to scheduling algorithms

to exploit the opportunity to saving energy, which leads to our investigation on scheduling a task graph on multiprocessor systems.

To the best of my knowledge, this dissertation is the first work to consider both throughput and response time constraints in energy-aware scheduling of task graphs, while there has been much research considering only response time constraint prior to this work. My main contribution is a novel scheduling algorithm called Scheduling2D that exploits the difference of the two QoS requirements to perform processor allocation, task mapping, and task speed scheduling simultaneously. The design of Scheduling2D emphasizes the use of multiple processors as an energy reduction technique because of the fundamental trade-off between dynamic and static energy consumption. The derivation of Scheduling2D shows that the static power of processors has an important impact on the scheduling of streaming applications. Specifically, the static power imposes an upper bound on the response time of the streaming application to be scheduled and high static power could lead to servicing requests faster than the response time requirement in order to save energy. This is contrary to the common DVS wisdom of slowing down task execution as much as possible for just-in-time completion. This important insight is also confirmed by the experimental results on Scheduling2D.

7.0 FUTURE WORK

Energy-aware scheduling for streaming applications is an exciting research area and the problems we considered in this dissertation can be extended in various directions. Next, I elaborate on the research avenues that I consider promising for future work in this area.

I obtained scheduling algorithms under the realistic processor model for all the problems considered. I believe that the power portion in the realistic model is very realistic in the sense that each task has individual power consumption for each discrete frequency. However, for the timing portion in the realistic model, I made an assumption that the data processing of a task is performed on fast local memory. Thus, each task is CPU-bound and its execution time is inversely proportional to its operating frequency. This greatly simplifies computing the execution time of a task in deriving scheduling algorithms. It is worthwhile to do research on the case where a task is I/O-bound (for example, the task needs to reference a lot of data residing in disks). To do that, one needs to strengthen the timing modeling in the realistic model and revisit all the problems.

Energy-aware scheduling of streaming applications on uniprocessor systems is relatively well understood. However, there is much room left for research in scheduling on multiprocessor systems. There are a number of directions for future work.

1. Morphing task graphs into a two-dimensional structure is very important for the Scheduling2D algorithm. Currently, a simple topological sort is employed for this step. Better heuristics are expected to be obtained by considering the computational requirement (i.e., cycle count) of each task.

2. For stochastic workload, I have shown in Section 5.3.3 the difficulty of incorporating dynamic slack reclamation into the offline part of Scheduling2D to compute the expected

energy consumption. The resulting solution for stochastic workload, SScheduling2D, is only a simple extension to Scheduling2D. Although the SScheduling2D algorithm can still guarantee the QoS requirements, it is not clear how far the resulting energy consumption is from the optimal solution. Whether or not one can directly optimize the expected energy consumption in the offline part of the algorithm needs further research.

3. Because this dissertation focuses on energy reduction, I only consider memory power consumption and implicitly assume unlimited memory size, which is not true especially for on-chip local memory or embedded systems that have stringent memory constraint. Adding memory size as an additional constraint to the scheduling problems has significant interest. I envision that the X-mapping of the Scheduling2D algorithm is the very place to deal with memory constraint.
4. All multiprocessor scheduling algorithms in this dissertation assume unlimited number of processors in the system based on the trend that more and more processor cores are available on chip multiprocessors. However, if multiple streaming applications are executed in the same system, we may be constrained by the number of processors. Given the already complex nature of multiprocessor scheduling problems, dealing with this constraint is expected to be a challenging problem.

Finally, the problems considered in this dissertation can be expanded by taking into account Dynamic Power Management (DPM) that attempts to put idle system components into low-power states (e..g, turning off) whenever possible. This dissertation only considers Static Power Management, which means once a scheduling algorithm decides that a processor should be on, it will never be turned off when executing streaming applications. Combining DVS and DPM has been shown to achieve further energy savings than DVS alone [73, 17]. How to incorporating DPM into the scheduling algorithms in this dissertation (especially for multiprocessor systems) is an interesting problem for future research.

APPENDIX A

AN ILLUSTRATIVE EXAMPLE OF SPEED ROUNDING EFFECT

In Section 4.3.1, we described how the GRACE and PACE schemes round the continuous speed obtained from the ideal processor model to comply with the realistic model. In this appendix, we demonstrate the speed rounding can have a significant impact on the quality of the solution through an illustrative example. In this example, there is a single task τ that has 3 cycles and its deadline is 1.84 time units. The processor has 3 discrete frequencies, that is, 1 Hz, 2 Hz, and 3Hz. The processor power model is $p(f) = f^3$ and speed change overhead is zero.

Suppose that the probability function of the execution cycles is $P(1) = 0.83, P(2) = 0.05, P(3) = 0.12$. We can compute the cumulative function and obtain $cdf(1) = 0.83, cdf(2) = 0.88, cdf(3) = 1$. The expected energy consumption, according to Equation (4.1), is

$$s_1^2 + 0.17 \times s_2^2 + 0.12 \times s_3^2$$

and the optimal continuous speed schedule, according to Equation (4.3), is $s_1 = 1.1126, s_2 = 2.0084, s_3 = 2.2557$. As described in Section 4.3.1, GRACE rounds a continuous speed up to the closest higher discrete frequency, while PACE rounds a continuous speed up or down to the closest discrete frequency. Thus, GRACE will use the speed schedule $s_1 = 2, s_2 = 3, s_3 = 3$ to execute the task and result in the expected energy consumption of 6.61. However, using the speed scheduling $s_1 = 1, s_2 = 2, s_3 = 3$ will give us the optimal expected energy consumption, which is 2.76. Therefore, GRACE will have the relative error (defined in Section 4.3.4.1) of $\frac{6.61-2.76}{2.76} = 139\%$. For PACE, the original speed schedule is

$s_1 = 1, s_2 = 2, s_3 = 2$, which will make the task miss the deadline since in the worst case the task will take time $\frac{1}{1} + \frac{1}{2} + \frac{1}{2} = 2$, which is greater than 1.84. PACE will perform a linear scan and adjust the speed schedule. The new speed schedule is $s_1 = 1, s_2 = 2, s_3 = 3$, which actually is the optimal speed schedule. Thus, the relative error of PACE is 0 in this case.

Suppose that the probability function of the execution cycles is changed to $P(1) = 0.96, P(2) = 0.02, P(3) = 0.02$. We compute the cumulative function and obtain $cdf(1) = 0.96, cdf(2) = 0.98, cdf(3) = 1$. Thus, the expected energy consumption according to Equation (4.1) is

$$s_1^2 + 0.04 \times s_2^2 + 0.02 \times s_3^2$$

and the optimal continuous speed schedule, according to Equation (4.3), is $s_1 = 0.8768, s_2 = 2.5639, s_3 = 3.2304$. For PACE, the speed schedule is $s_1 = 1, s_2 = 3, s_3 = 3$ and the resulting expected energy consumption is 1.54. However, the optimal speed schedule is still $s_1 = 1, s_2 = 2, s_3 = 3$ and the optimal expected energy consumption is 1.34. Therefore, PACE will have the relative error of 15%. For GRACE, the speed schedule is the same as that of PACE and thus GRACE also has the relative error of 15%.

APPENDIX B

AN ILLUSTRATIVE EXAMPLE OF DVS SCHEMES

In this appendix, we demonstrate and compare several DVS schemes for general frame-based systems under the ideal processor model through an illustrative example. In this example, there are 3 tasks in a frame-based real-time system with a frame length of 14 time units. The parameters for the 3 tasks are shown in Table 9. The tasks are required to be executed in the order of τ_1, τ_2, and τ_3. We can also treat the 3 tasks as three sequential sections of a single task $\hat{\tau}$ and its parameters are computed from those of the 3 tasks. $\hat{\tau}$ is used for a naive extension of PACE shown at the end of this appendix. The processor power model is $p(f) = f^3$.

We first look at a simple scheme called the Proportional scheme [48], which distributes the system slack proportionally among all unexecuted tasks. In this example, the Proportional scheme will start executing τ_1 using speed $\frac{2+4+2}{14} = 0.5714$. After τ_1 finishes, the system reclaims the slack created by τ_1 if it runs for less than its WCEC, and computes the speed of the next task recursively. Suppose that τ_1 only runs for 1 cycle. Then the time left for executing τ_2 and τ_3 is $14 - \frac{1}{0.5714} = 12.2499$, and speed $\frac{4+2}{12.2499} = 0.4898$ will be used to execute τ_2, and so forth.

For the OITDVS scheme, the time allocation fractions of τ_1, τ_2, τ_3 are $\beta_1 = 0.3938, \beta_2 = 0.7619$, and $\beta_3 = 1.0$, respectively (refer to Algorithm 4.1 for how to compute the β values). Thus, OITDVS will use speed $\frac{2}{0.3938 \times 14} = 0.3628$ to execute τ_1. If τ_1 runs for 1 cycle, then the time left for executing τ_2 and τ_3 is $14 - \frac{1}{0.3628} = 11.2737$. Then OITDVS will use speed $\frac{4}{0.7619 \times 11.2737} = 0.4657$ to execute τ_2.

Table 9: The parameters for the 3 tasks in the illustrative example

Task	W	$P(1), P(2), \cdots, P(W)$
τ_1	2	.9, .1
τ_2	4	.9, 0, 0, .1
τ_3	2	.5, .5
$\hat{\tau}$	8	0, 0, .405, .45, .045, .045, 0.05, .005

For the GOPDVS scheme, the time allocation fractions of τ_1, τ_2, τ_3 are $\beta_{11} = 0.2147, \beta_{12} = 0.2207, \beta_{21} = 0.2832, \beta_{22} = 0.2086, \beta_{23} = 0.2636, \beta_{24} = 0.3579, \beta_{31} = 0.5575$, and $\beta_{32} = 1.0$, respectively (refer to Algorithm 4.2 for how to compute the β values). Thus, GOPDVS will use speed $\frac{1}{0.2147 \times 14} = 0.3327$ to execute the 1^{st} cycle of τ_1. If τ_1 has the 2^{nd} cycle, the GOPDVS scheme will use speed $\frac{1}{0.2207 \times (14 - 14 \times 0.2147)} = 0.4121$ to execute. Table 10 shows the expected energy consumption per frame for the DVS schemes.

Table 10: The comparison of the DVS schemes for the illustrative example

Scheme	Expected energy consumption per frame
naive PACE	0.7953
Proportional	0.7733
OITDVS	0.6097
GOPDVS	0.5154

Finally, we show through this simple example that a naive extension of PACE (or, naive PACE for short) cannot even obtain energy savings over the DVS schemes that do not use intra-task DVS. Since PACE has only been studied for a single task, naive PACE is applied to the supertask $\hat{\tau T}$ in Table 9. From Table 10, we can see that naive PACE [40] will result in expected energy consumption per frame of 0.7953, which is even worse than the Proportional scheme.

APPENDIX C

PROOF OF LEMMA 2

In this appendix, we provide the proof of Lemma 2, which states that for every energy-time label $l' \in LABEL'(i,j)$ where $1 \leq j \leq M$, there exists a label $l \in LABEL(i,j)$ such that $l'.e \leq l.e \leq (1+\delta)^i l'.e$ and $l'.t \geq l.t$.

Proof. We prove by induction on i. The base case ($i = 0$) is trivially true.

In the inductive step, we assume that the claim holds for i. Now the goal is to find an energy-time label $\zeta \in LABEL(i+1,j)$ $(1 \leq j \leq M)$ for every energy-time label $l'_{i+1} \in LABEL'(i+1,j)$ such that

$$l'_{i+1}.e \leq \zeta.e \leq (1+\delta)^{i+1} l'_{i+1}.e$$

and

$$l'_{i+1}.t \geq \zeta.t$$

Let $l'_i \in LABEL'(i,k)$ $(1 \leq k \leq M$, and k is not necessarily equal to $j)$ be the energy-time label that generates l'_{i+1}. Then we have

$$l'_{i+1} = (l'_i.e + P_C(i)P_E(f_k,f_j) + F_i e(f_j),\ l'_i.t + P_T(f_k,f_j) + \frac{w_i}{f_j})$$

By the induction hypothesis, there is an energy-time label $l_i \in LABEL(i,k)$ such that

$$l'_i.e \leq l_i.e \leq (1+\delta)^i l'_i.e$$

and

$$l'_i.t \geq l_i.t$$

Let $l_{i+1} = (l_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j),\ l_i.t + P_T(f_k, f_j) + \frac{w_i}{f_j})$. Thus, we have

$$l_{i+1}.e < (1+\delta)^i l'_i.e + (1+\delta)^i (P_C(i)P_E(f_k, f_j) + F_i e(f_j)) = (1+\delta)^i l'_{i+1}.e$$

There are two possibilities regarding l_{i+1}:

1. $l_{i+1} \in LABEL(i+1, j)$. Then we have

$$l'_{i+1}.e = l'_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j) \leq l_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j) = l_{i+1}.e$$

and

$$\begin{aligned} l_{i+1}.e &= l_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j) \\ &\leq (1+\delta)^i l'_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j) \\ &\leq (1+\delta)^i (l'_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j)) \\ &= (1+\delta)^i l'_{i+1}.e \\ &< (1+\delta)^{i+1} l'_{i+1}.e \end{aligned}$$

and

$$l'_{i+1}.t = l'_i.t + P_T(f_k, f_j) + \frac{w_i}{f_j} \geq l_i.t + P_T(f_k, f_j) + \frac{w_i}{f_j} = l_{i+1}.t$$

Therefore we let $\zeta = l_{i+1}$.

2. $l_{i+1} \notin LABEL(i+1, j)$ and it was removed as a result of trimming. Thus there is an energy-time label $\hat{l} \in LABEL(i+1)$ such that

$$l_{i+1}.e \leq \hat{l}.e \leq (1+\delta) l_{i+1}.e$$

and

$$l_{i+1}.t > \hat{l}.t$$

Therefore we have

$$l'_{i+1}.e = l'_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j) \leq l_i.e + P_C(i)P_E(f_k, f_j) + F_i e(f_j) = l_{i+1}.e \leq \hat{l}.e$$

and

$$\hat{l}.e \leq (1+\delta)l_{i+1}.e < (1+\delta)(1+\delta)^i l'_{i+1}.e \leq (1+\delta)^{i+1} l'_{i+1}.e$$

and

$$l'_{i+1}.t \geq l_{i+1}.t > \hat{l}.t$$

Therefore we let $\zeta = \hat{l}$.

Summarizing the above two possibilities will prove the claim. □

BIBLIOGRAPHY

[1] Cell broadband engine architecture documentation, 2005.

[2] Intel developer forum, 2006. http://www.intel.com/pressroom/kits/events/idffall_2006 /pdf/idf 09-26-06 paul otellini keynote transcript.pdf.

[3] N. AbouGhazaleh, D. Mossé, B. Childers, R. Melhem, and Matthew Craven. Collaborative operating system and compiler power management for real-time applications. In *Proc. IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, May 2003.

[4] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: A scheduling heuristic for streaming applications on the grid, 2005.

[5] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. Al-Hashimi. Simultaneous communication and processor voltage scaling for dynamic and leakge energy reduction in time-constrained systems. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, 2004.

[6] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[7] H. Aydin. *Enhancing Performance and Fault Tolerance in Reward-Based Scheduling.* PhD thesis, University of Pittsburgh, 2001.

[8] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 95–105, December 2001.

[9] S. Baruah and J. Anderson. Energy-efficient synthesis of periodic task systems upon indentical multiprocessor platforms. In *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 428–435, Tokyo, Japan, 2004.

[10] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Bi-criteria pipeline mappings for parallel image processing. In *Proc. 8th International Conference on Computational Science (ICCS)*, LNCS. Springer Verlag, 2008.

[11] V. Bharadwaj, D. Ghose, and T.G.Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6:7–18, 2003.

[12] Shahid H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, 1987.

[13] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4), 1999.

[14] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-aware Microarchitecture: Design and Modeling Challenges for Next Generation Microprocessors. *IEEE Micro*, 20(6), 2000.

[15] T. Burd and R. Brodersen. Design issues for Dynamic Voltage Scaling. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, June 2000.

[16] J.J. Chen, H.R. Hsu, and T.W. Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, 2006.

[17] J.J. Chen and L. Thiele. Energy-efficient scheduling on homogeneous multiprocessor platforms. In *Proc. ACM Symposium on Applied Computing*, Sierre, Switzerland, 2010.

[18] J. Cong and K. Gururaj. Energy efficient multiprocessor task scheduling under input-dependent variation. In *Proc. Design, Automation and Test in Europe*, Dresden, Germany, 2009.

[19] G. Contreras and M. Martonosi. Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, August 2005.

[20] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, 1990.

[21] D. Duarte and N. Vijaykrishnan and M. J. Irwin and H-S Kim and G. McFarland. Impact of scaling on the effectiveness of dynamic power reduction schemes. In *Proc. International Conference on Computer Design (ICCD)*, 2002.

[22] E.N. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proc. Workshop on Power-Aware Computer Systems (PACS)*, 2002.

[23] A. Elyada, R. Ginosar, and U. Weiser. Low-complexity policies for energy-performance tradeoff in chip-multi-processors. *IEEE Transactions on Very Large Scale integration systems*, 16(9), 2008.

[24] F. Gruian and K. Kuchcinski. Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2001.

[25] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[26] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002.

[27] F. Gruian. Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS Processors. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, August 2001.

[28] F. Gruian. On Energy Reduction in Hard Real-Time Systems Containing Tasks with Stochastic Execution Times. In *Proc. IEEE Workshop on Power Management for Real-Time and Embedded Systems*, Taipei, Taiwan, May 2001.

[29] F. Gruian and K. Kuchcinski. Uncertainty-Based Scheduling: Energy Efficient Ordering for Tasks with Variable Execution Time. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, Seoul, Korea, August 2003.

[30] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998.

[31] I. Ahmad and Y.K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel and Distributed Systems*, 9(9), 1998.

[32] T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202, August 1998.

[33] N. S. Kim, T. Kgil, K. Bowman, V. De, and T. Mudge. Towards power-optimal pipelining and parallel processing under process variations in nanometer technology. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, 2005.

[34] W. Kim, D. Shin, H. Yun, J. Kim, and S. Min. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In *Proc. IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, 2002.

[35] S. Krantz, S. Kress, and R. Kress. *Jensen's Inequality*. Birkhauser, 1999.

[36] S. Lang. *Calculus of Several Variables*. Addison-Wesley, 1973.

[37] P.D. Langen and B. Juurlink. Trade-offs between voltage scaling and processor shutdown for low-energy embedded multiprocessors. *Lecture Notes in Computer Science*, 4599, 2007.

[38] W.Y. Lee. Energy-saving dvfs scheduling of multiple periodic real-time tasks on multi-core processors. In *Proc. 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applicatio*, Singapore, 2009.

[39] J. Lorch. *Operating Systems Techniques for Reducing Processor Energy Consumption.* PhD thesis, University of California at Berkeley, 2001.

[40] J. Lorch and A. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proc. ACM SIGMETRICS*, June 2001.

[41] J. Lorch and A. Smith. Operating system modifications for task-based speed and voltage scheduling. In *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, May 2003.

[42] J. Lorch and A. Smith. PACE: a New Approach to Dynamic Voltage Scaling. *IEEE TRansactions on Computers*, 53(7):856–869, 2004.

[43] A. Mahalanobis, B. V. K. Vijaya Kumar, and S.R.F. Sims. Distance-classifier Correlation Filters for Multiclass Target Recognition. *Applied Optics*, 35, 1996.

[44] Steven Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, 2002.

[45] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem. Energy aware scheduling for distributed real-time systems. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, 2003.

[46] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical Power Slope: Understanding the runtime effects of Frequency Scaling. In *Proc. ACM International Conference on Supercomputing*, June 2002.

[47] B. Mochocki, X. Hu, and G. Quan. A Unified Approach to Variable Voltage Scheduling for Nonideal DVS Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1370–1377, September 2004.

[48] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-Assisted Dynamic Power-aware Scheduling for Real-Time Applications. In *Proc. Workshop on Compiler and OS for Low Power (COLP)*, October 2000.

[49] M.T. Yang and R. Kasturi and A. Sivasubramaniam. A pipeline-based approach for scheduling video processing algorithms on now. *IEEE Trans. Parallel and Distributed Systems*, 14(2), 2003.

[50] P. D. Hoang and Jan M. Rabaey. Scheduling of dsp programs onto multiprocessors for maximum throughput. *IEEE Trans. Signal Processing*, 41(6), 1993.

[51] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102, October 2001.

[52] F. Preparata and M. Shamos. *Computational Geometry An Introduction.* Springer, 1993.

[53] R. P. Dick and D. L. Rhodes and W. Wolf. Tgff: Task graphs for free. In *Proc. Sixth International Workshop on Hardware/Software Codesign*, 1998.

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

[55] S. Banerjee and T. Hamada and P. M. Chau and R. D. Fellman. Macro pipelining based scheduling on high-performance heterogeneous multiprocessor systems. *IEEE Trans. Signal Processing*, 43(6), 1995.

[56] S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *Proc. IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, May 2003.

[57] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[58] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Proc. International Conference on Compiler Construction*, Grenoble, France, 2002.

[59] H. S. Wang, X. Zhu, L. S. Seh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proc. International Symposium on Microarchitecture*, 2002.

[60] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, MA, 1993.

[61] H. Lee H. Kim W.Y. Lee, Y.W. Ko. Energy-efficient scheduling of a real-time task on dvfs-enabled multi-cores. In *Proc. International Conference on Hybrid Information Technology*, Seoul, Korea, 2009.

[62] C. Xian, Y.H. Lu, and Z. Li. Energy-aware scheduling for real-time multiprocessor systems with uncertain task. In *Proc. Design Automation Conference (DAC)*, San Diego, CA, 2007.

[63] F. Xie, M. Martonosi, and S. Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

[64] Intel XScale Microarchitecture:Benchmarks, 2005. http://web.archive.org/web/20050326232506/http://developer.intel.com/design/intelxscale/benchmarks.htm.

[65] R. Xu, R. Melhem, and D. Mossé. A Unified Approach to Stochastic DVS Scheduling. In *Proc. of ACM International Conference on Embedded Software (EMSOFT)*, Salzburg, Austria, October 2007.

[66] R. Xu, R. Melhem, and D. Mossé. Energy-Aware Scheduling for Streaming Applications on Chip Multiprocessors. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, Tucson, AZ, December 2007.

[67] R. Xu, D. Mossé, and R. Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Transactions on Computer Systems (TOCS)*, 25(4), 2007.

[68] R. Xu, C. Xi, R. Melhem, and D. Mossé. Practical PACE for Embedded Systems. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*, Pisa, Italy, September 2004.

[69] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé. Energy-efficient policies for embedded clusters. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2005.

[70] Y. Chow and F. Anger and C. lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Computers*, 18(2), 1989.

[71] W. Yuan and K. Nahrstedt. Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[72] Y. Zhang, Z. Lu, J. Lach, K. Skadron, and M. Stan. Optimal Procrastinating Voltage Scheduling for Hard Real-Time Systems. In *Proc. Design Automation Conference (DAC)*, June 2005.

[73] B. Zhao and H. Aydin. Minimizing expected energy consumption through optimal integration of dvs and dpm. In *Proc. International Conference on Computer-Aided Design*, San Jose, CA, 2009.

INDEX