ABSTRACT

Energy-Aware Operating System Design

by

Padmanabhan Pillai

Chair: Kang G. Shin

As computer technology progresses, making smaller, lighter, and faster computing devices possible, we are experiencing a proliferation of high-performance handheld and mobile, embedded computing devices. As these devices become faster, applications more complex, and microprocessors more powerful, it is becoming difficult to provide sufficient runtime with the available stored energy in small devices. To address this, research is now starting to look into low-power and energy-conserving computer designs. Much of this work is in the domain of microprocessor and system hardware designs that reduce the energy required for computation, generally involving a trade-off between computational throughput and power consumption.

This thesis considers a software-centric approach to designing low-power embedded and mobile computing systems, identifying what must be done in the system software to create an energy-aware and -efficient system. A three-pronged approach of designing an energy-conserving OS is proposed, focusing primarily on reducing processing energy costs.

- First, we survey various techniques of optimizing OS services to the expected characteristics of small, embedded systems, in order to reduce processing and energy overheads for the common-case scenarios.

- Second, algorithms for real-time task scheduling are devised in order to exploit energy-conserving hardware techniques, while alleviating their performance impacts. Sprint-and-halt algorithms and real-time dynamic voltage scaling (RT-DVS) algorithms are developed to exploit software-controlled power down of hardware, and processor voltage and frequency scaling, respectively, to reduce energy consumption in real-time systems while maintaining all deadline guarantees.

- Third, a comprehensive system of task-adaptation, called the Energy-aware Quality-of-Service (EQoS) framework is introduced. This framework provides differentiated service quality to applications tasks, weighing energy requirements against value of computation and system runtime, in order to maximize the total value of computation in a time- and energy-constrained environment.

In addition, as the energy requirements of applications is a critical input to any energy-based adaptation system, methods of measuring and modeling task energy requirements are also explored. This work investigates energy-aware operating system design by developing and implementing software mechanisms and algorithms in working systems, demonstrating the efficacy of the energy-conserving techniques developed.

Energy-Aware Operating System Design

by

Padmanabhan Pillai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

Doctoral Committee:
Professor Kang G. Shin, Chair
Assistant Professor Brian Noble
Associate Professor Steven Reinhardt
Associate Professor Dawn Tilbury

To my wife and my parents.

ACKNOWLEDGEMENTS

This thesis was possible only with the support and encouragement of many people, to whom I am deeply grateful.

I am most indebted to my wife, Deepa, who was there to support me when I needed it the most. She had to tolerate my long hours during the actual writing phase of my thesis work, and selflessly took this without complaint. She never let me lose sight of my goal, encouraging me to persevere.

I am thankful to my parents and my brother, Kamal, who supported me from the start. From an early age, my parents instilled in me the desire to pursue my education to the fullest and to strive for success. They supported and accepted my decision to enter and stick with a Ph.D. program.

My advisor, Professor Kang Shin, has been instrumental in the success of this work. He gave me full freedom to pursue any interesting avenue of research, but directed me always toward high quality research directions.

I would like to thank the other members of my thesis committee, Professors Brian Noble, Steven Reinhardt, and Dawn Tilbury, for their comments and insights that helped shape and clarify the early direction of this thesis.

Finally, I would like to thank Cheng Jin, Sung-Whan Moon, Hai Huang, Hani Jamjoom, and John Reumann for their roles as collaborators, technical sounding boards, and, mostly, for just being good friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

Computer use has been evolving continuously as new technologies develop and as computers both integrate into and change society as a whole. During the earliest days of commercial computing, computer use was very restricted, primarily consisting of batch processing tasks and accounting on large, off-site mainframe machines. Later, the advent of mini-computers permitted computer use to become more widespread and interactive through timesharing systems. This trend of smaller, cheaper, more-widely deployed computer systems continued through the 80's and 90's with the introduction and subsequent spread of personal computer technology to businesses and homes. This general trend, pushed by the industry's fixation on Moore's Law and exponentially-improving technology, shows no sign of slowing. In this decade, we are seeing the results of this — a further spread of computer use as mobile computing devices of all sorts become ubiquitous.

Many technologies are enabling this explosion of mobile and hand-held computing devices. Wireless networks are improving and becoming more widespread in use, both as local-area networks (LANs), such as WaveLAN, and in wide-area networks (WANs), particularly through digital cellular networks. Ever-improving integration technology is providing low-cost system-on-a-chip solutions for mass-produced consumer computing devices. Improved, miniaturized storage systems, both solid state (Flash memory), and magnetic (1-inch hard drives), are providing mass-storage solutions for mobile devices in very small physical form factors.

Furthermore, these technologies enable new applications and shifts in usage patterns, increasing the demand for mobile computing devices. Low-cost CCD sensors and Flash memory devices have enabled mass-market digital cameras that are now rapidly displacing traditional film cameras as the preferred photographic medium. These incorporate sophisticated embedded computers to acquire, process, compress, store, and manage digital images through a user-friendly interface. Mobile computing platforms, including laptops and hand-held devices, are important productivity enhancers for business travelers. PDAs and similar devices with wireless communication are now indispensable to many businesses that rely on continuous e-mail connectivity. Mobile phones had been primarily limited to voice communications, but with integration into wireless digital data networks, are now capable of instant person-to-person messaging, e-mail, and photo messaging. As wireless networks improve in availability, bandwidth, and Internet connectivity, these networks may be able to leverage upon existing service infrastructure to provide a wide array of information and entertainment content to mobile clients. Entertainment services, in particular, could be popular on hand-held devices, as online gaming, music and photo sharing, and multimedia streaming applications available to PCs trickle down to smaller platforms as the devices become more powerful and wireless networking improves.

Despite these motivators for mobile systems usage, there are a few potential stumbling blocks on the way toward widespread deployment. Chief among these is the issue of providing the energy needed for high-performance mobile computation. As applications and platforms become more complex and require greater computational capacity, it becomes increasingly difficult to provide sufficient energy to perform such computation in a mobile environment. As the performance of mobile processors increase, so do their energy requirements, quickly dominating system energy requirements. Even in notebook computer systems that have significant external components to the processor, such as large, backlit displays and spinning mechanical hard drives, peak power is dominated by the processor. In smaller mobile devices, such as PDAs that have fewer external components, energy consumption is even more dependent on the processor characteristics. When the size and weight of a device are constrained, increasing computational capacity increases total energy consumption and reduces battery life. Although battery technologies have been improving, the exponential increase in processor and software complexity has far out-paced the relatively slow improvements in energy storage technology.

As a result, there is a great need for technologies that can reduce energy consumption in mobile devices while maintaining high peak performance for the increasingly complex and computationally intensive applications demanded in next-generation products. Much research and development is now focused on reducing energy consumption, particularly through low-power hardware development. Improved semiconductor process technologies, greater integration of parts, reduced voltage circuits, and gated clock drivers are all contributing to improved energy efficiency in handheld devices. Other hardware approaches involve low-power states and components that trade speed for reduced energy consumption and corresponding improvement in battery life. Although such techniques can conserve considerable amounts of energy, in general, they cause some degradation in performance and affect task execution timings. Particularly in embedded mobile devices that must provide robust user interaction or perform computations with real-time constraints, any performance degradation or timing change may result in poor or even incorrect application behavior.

This thesis takes a complementary, software-centric approach to designing systems for low energy consumption. In particular, mechanisms of improving the efficiency of system services and better control of hardware-based power conservation mechanisms in the operating system are considered. As the processor tends to be the largest single consumer of energy in computational devices, the approaches in this thesis tend to focus on processing related energy consumption. The software approach taken here ensures that any performance impacts of energy-conserving mechanisms are mitigated, avoiding timing violations in real-time embedded systems. Furthermore, the software approach to energy-efficient systems permits intelligent allocation of limited energy resources to different applications based on high-level knowledge of the system and application goals, in order to make the best use of the available energy. Such differentiation between tasks and distribution of energy resources cannot be achieved through hardware mechanisms alone. The remainder of this chapter outlines the various aspects of a software-centric approach to low-power system design.

1.1 Software Approach to Energy Reduction

Energy availability is a primary limitation in most mobile and portable devices. To handle the increasingly complex applications that are rapidly emerging in a highly competitive consumer marketplace, devices are being developed with faster, higher-power processors and larger memories, increasing the power burden in such systems. Unfortunately, with the available battery technologies, the size and weight constraints limit systems either to to low-performance components that dissipate low power, or to shortened battery life. In order to maintain acceptable battery life as handheld devices evolve into higher performance platforms, mechanisms to improve the energy efficiency of systems are required. In this work, a software-centric approach to improving energy efficiency in handheld devices is explored.

Of course, energy is not the only constraint that software must contend with in mobile devices. As the applications become increasingly feature-rich, managing system resources and dealing with the interactions between multiple, sophisticated application tasks executing in parallel becomes a complex task. To deal with this complexity, mobile and portable devices often need full operating systems rather than ad-hoc, in-application mechanisms used in simpler systems. In addition, these systems often have strict timing constraints, requiring real-time operating systems to ensure deadlines are met for all application tasks. For example, a digital voice communication application would need to acquire, compress, and packetize each frame of audio within a fixed deadline in order to ensure no gaps or jitter in the transmitted audio. A digital camera may have very strict timing requirements for the task that actually controls the picture taking operation to ensure correct exposures and proper acquisition of CCD data, while image compression and user interface tasks do not need such strict constraints, and only need to be sufficiently responsive to a human user. Hence, software techniques for power reduction must ensure that performance and real-time constraints are maintained.

This work seeks to develop techniques of improving energy consumption in embedded system through the development of energy-aware operating systems. The primary focus is on the system software, i.e., the operating system, as this is responsible for managing all resources in the system, and energy should be no exception. Improving the OS and making it energy-aware will allow benefits to be spread to all applications running on the system, and not limited to a particular single task.

To this end, this thesis takes a 3 pronged approach to software-centric energy conservation. First, improving the processing efficiency of OS itself is considered. Reducing the energy overheads of OS services will improve the energy efficiency of all applications that use these services. Such improvements may preserve the semantics of the original service, or can take advantage of characteristics unique to the embedded or real-time system to gain in terms of energy reductions.

Hardware techniques that slow or power down system components can conserve significantly more power than these software optimizations alone can achieve. However, these techniques all incur some form of performance penalty, and may cause timing failures in real-time systems. The second aspect of an energy-aware OS is to effectively control and maximally exploit hardware power-reduction technology such that it does not interfere with the system performance, particularly any real-time or execution time constraints. As the different hardware techniques employ varying mechanisms to conserve energy, the algorithms and software techniques to control and maximize the energy benefits will vary with the type of hardware involved.

Finally, the system software needs to manage and allocate energy resources to the most beneficial tasks in order to make best use of the limited resource. In order to achieve this, a system of adapting the working task set to the available energy is proposed. By selectively varying the level of service quality provided to each task, the OS can weigh the benefits gained from an application against the energy consumption of the tasks in a general, energy-aware framework.

1.2 Improving OS Service Efficiency

From a high-level perspective, each unit of computation time on a processor incurs some energy cost, regardless of the purpose of the computations. The less that a function does, the lower the energy consumed in executing that function. If an OS service can provide its functionality with lower overheads, or provide a reduced service with lower computational requirements, this will free energy and computation resources that can be used for other computations. By reducing the overheads in an OS service, all tasks using the service will consume less energy, allowing more total useful work to be accomplished with the same amount of energy.

The most direct approach is to cut the excess overheads from a service while maintaining all service semantics. This will allow applications to remain unmodified and ensure no degradation occurs within the applications to make up for the missing or changed service behaviors. In general, most performance enhancing techniques will also result in reductions in energy, since the reduction in the number of computation cycles used directly lowers the energy consumed. Hence, almost any technique that can reduce the processing time of a service call can be applied to also reduce the energy consumed in the service. The exception to this is the case where performance is improved through speculative prefetching or by working ahead. Here, extra computation is done in the hope that the results will be needed later. If so, this improves performance as the computations are completed early. However, there is also the chance that the work is unnecessary, or based on incorrectly speculated data, in which case the results must be discarded, wasting the energy resources consumed in the speculative execution.

In general, this direct approach of reducing overheads to improve energy consumption is fairly limited, as performance improvements in OS services is a well-investigated topic, and many commercial operating systems already provide low-overhead services. A second approach is to modify the semantics of the services to take advantage of known characteristics of the embedded or real-time system to reduce service overheads and improve energy conservation. As the operation of the OS services is changed, from handling a broad general-case scenario, to the specific cases likely on a small, handheld device, there is a potential for significant overhead reductions. Certain aspects of small embedded devices can be used to revamp OS services for reduced processing overheads and energy consumption. For example, the various tasks executing on embedded platforms tend to be static or change infrequently, and also tend to be well-specified, rather than arbitrary applications. In such a situation, eliminating naming services, extra layers of indirection, and table lookups needed in more dynamic environments, and instead relying on statically-specified resources can reduce the overheads of many services, particularly those dealing with shared memory or semaphores. With real-time scheduling, timing constraints of different tasks can be used to reduce the need for strict synchronization in some services, such as interprocess communication, which can use the guaranteed temporal characteristics of the tasks to ensure correct behavior. Revising the OS service semantics to take advantage of the unique characteristics of embedded, mobile systems can result in services with greatly reduced processing and energy overheads.

These general approaches can conserve energy only if the time gained from reduced service overheads is used for some valuable computation. If the excess time is spent on idle loops, then the energy saved in OS services would be wasted in executing the idle loops, and total energy consumption not significantly improved. On the other hand, if the cycles are spent in useful work, then although the total energy expenditure may not be affected, the energy efficiency, in terms of energy per unit of application computation, is still improved, and more valuable work is achieved from a given starting energy level. Finally, if the platform can be powered off or switched to a low-power state earlier, or for a longer duration, due to the reduced OS service overheads, then both the efficiency of application computation and total energy expenditure can be improved. This last feature can only be used if the system software appropriately uses hardware mechanisms to power-down the system in a way that preserves timeliness and responsiveness of the system.

1.3 Exploiting Hardware Energy-Conservation Mechanisms

The software optimizations discussed so far reduce energy costs for tasks by reducing the total number of instructions executed by OS calls and services. These, in turn, reduce the energy costs and execution time for the tasks using the OS services, resulting in a surplus of energy and processor time. When this surplus is used to perform other useful work, the energy cost per unit of useful computation is reduced, thus improving energy cost-effectiveness of computation.

There are two drawbacks with a software-only approach. First, although cost per unit computation is reduced, the total energy-consumption rate (i.e., power in Watts) of the device remains constant, so the total duration of operation with a given amount of energy is not improved. Second, if the surplus energy and processor cycles are not spent for useful work, then they will be wasted on idle loops. With some hardware support, however, both the energy cost per unit computation and the total energy consumption can be reduced.

Hardware energy-conservation techniques generally take one of two forms: they allow the system to turn off or put into low-power states the processor and other system components; or, they modify the operating characteristics of the device in order to reduce the energy dissipated. Both of these have some impact on performance — the former due to time and energy overheads of switching to and from low-power states, while in the latter, this is because the low-power operation generally comes with a cost in terms of processing speed. Due to the performance impacts, the system software needs to carefully manage the hardware to ensure applications continue to perform correctly, particularly in real-time embedded systems, where execution deadlines are must be met. The methods and algorithms needed to properly control and maximize the energy gains from the hardware mechanisms, while preserving real-time performance depends greatly on the nature of the hardware energy-conservation mechanisms.

There are two basic approaches to hardware energy conservation considered in this dissertation:

- *Software-controlled power-down*: the processing core and/or various system components can be switched to a low-power, inactive state by system software;

- *Processor frequency and voltage scaling*: the operating clock frequency and voltage of the processor are dynamically changed to reduce energy consumption.

Both mechanisms can save considerable amount of processing energy, but must be properly controlled by the operating system to ensure real-time tasks continue to receive guaranteed execution before their deadlines and performance of the system is not hampered by the energy conserving mechanisms.

1.3.1 Software-Controlled Power-Down

The basic idea behind software-controlled power-down is to deactivate the system when not performing useful work. During the execution of application tasks, there are often idle periods for various reasons. In interactive or networked devices, these occur commonly due to waiting for user input or some remote communication. Extra idle times are also inserted into schedules of real-time systems to keep some processor capacity in reserve to handle sporadic and aperiodic tasks triggered by external events. Furthermore, real-time tasks are scheduled assuming maximum, worst-case execution times (WCETs) to guarantee deadlines in all situations, but tasks typically take much less than the WCETs, resulting in idle time. Finally, surplus processing capacity is created as a result of the OS optimizations discussed earlier. During these idle times, the system should be powered down instead of wasting energy on idle loops.

In the simplest form of software-controlled power-down, the microprocessor has support for some form of a `halt` instruction. When executed, this causes the processor to stop the core, leaving it in an inactive state that reduces energy consumption. During any gap between useful computations, the `halt` instruction can be executed, conserving the energy that would otherwise be squandered in idle loops. The processor remains inactive until some form of interrupt, through some external event or a periodic timer tick, reactivates it, resuming normal processing. Ideally, the processor halt requires very little overhead — the processor halts and resumes nearly instantaneously (just a few cycles), so real-time task timings will not be affected. Because of the low overheads, it is trivial to employ processor halts at any detected idle period to conserve energy without affecting execution timings of real-time applications.

More generally, the system may have some mechanism of powering down the processor as well as external components under software control. Timers, video displays, and communication ports may be powered down to save considerable energy when not in active use. In particular, turning off the system clock-generation circuitry will essentially shut off the processor and memory, and often any video and communication subsystems as well. As a much larger piece of the system than just the processor is put into a low-power state, the potential savings may be much higher than with the processor halt alone. However, unlike the simple processor halt, powering down these subsystems incurs substantial overheads. Switching power-states may require several milliseconds, such as in waiting for the clock circuitry to stabilize on power-up. During the switch, neither useful work nor energy savings is realized. As a result, power-down of subsystems should be done only when the duration of power-down state is long enough to compensate for power-state switching overheads.

In order to make best use of such high-overhead, low-power mechanisms, the operating system scheduling policies must be modified to maximize the energy savings, while minimizing the total switching overheads incurred. In particular, a *sprint-and-halt* scheduling policy, where tasks are executed together, as fast as possible in a sprint mode, followed by a long halt period, formed by coalescing the normally occurring idle periods into longer segments, can maximize the power-down time and reduce the total overheads by incurring fewer power-state switches. Figure 1.1(a-c) illustrates task scheduling and power consumption with simple processor halts and sprint-and-halt scheduling. The main constraint on sprint-and-halt techniques is that the actual task timings are affected. A part of this dissertation seeks to develop scheduling techniques that maintain real-time scheduling and execution guarantees while simultaneously implementing sprint-and-halt algorithms to conserve energy and maximally exploit software-controlled power-down mechanisms.

1.3.2 Voltage and Frequency Scaling

The power-down approach has just two states of operation for the processor: full speed and halted. With appropriate hardware support, it is possible to vary the processor speed to achieve energy savings. *Frequency scaling* involves changing the frequency of the clock that drives the processor under software control. Since the energy consumed is proportional to the number of cycles executed, the OS can reduce power dissipation by throttling the clock rate. The speed of the processor should be set such that there is just enough computing capacity to complete the current tasks with no surplus, eliminating any wasteful idle loops. As the energy used in actual computation is unchanged, frequency scaling, at most, can only eliminate the processor energy consumed in idle loops, and cannot be expected to perform any better than processor halts. Alone, therefore, it is not very useful, but when used in conjunction with voltage scaling, it can reduce energy costs considerably.

Voltage scaling techniques use a software-controlled voltage regulator to adjust the operating voltage of the processor. Processor voltage greatly affects power dissipation, because static CMOS logic, the technology used in almost all contemporary microprocessors, behaves electrically like a switched capacitor. Each cycling of a gate input effectively charges and discharges a capacitor, consuming the stored energy which is proportional to voltage squared ($E = CV^2/2$) [6]. Hence, energy per machine cycle (and therefore

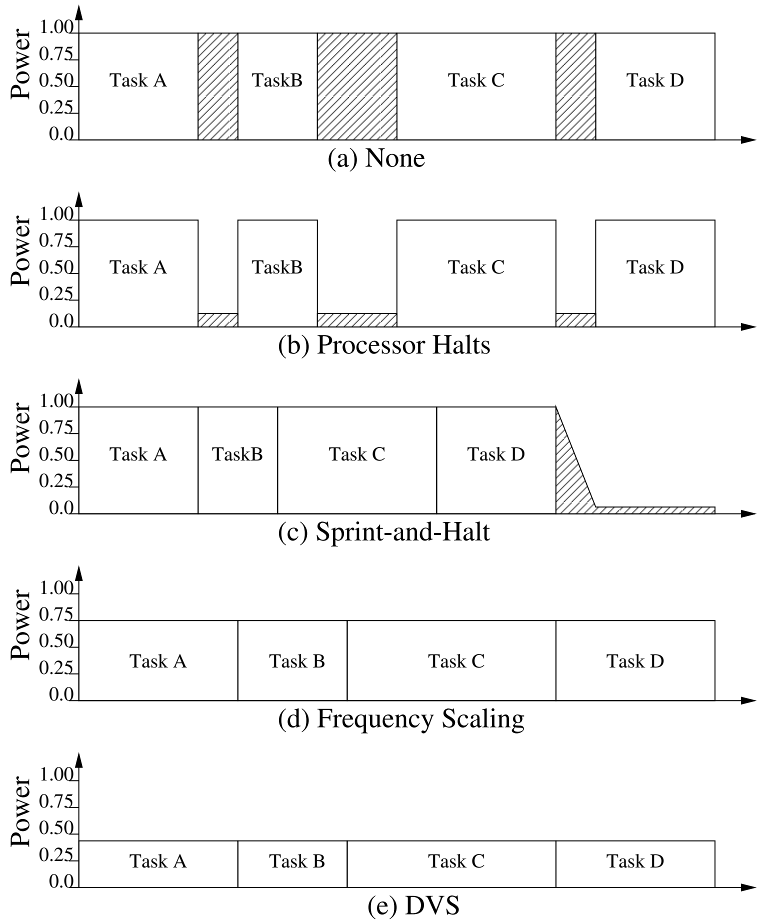


Figure 1.1: Illustration of energy savings with algorithms taking advantage of power conserving hardware. Shaded area is wasted energy that is not spent on useful computation.

per instruction) decreases with the square of the reduction in voltage, so, by employing a software-controllable voltage regulator to reduce the processor supply voltage, a large savings in energy consumption for any particular computational task may be realized.

Unfortunately, this does not come for free. As voltage is decreased, *gate delay* – the time it takes the output of a logic gate to reflect changes to the inputs – increases, with a relationship approximated by $D \propto V/(V - C)$, for some constant C [61]. Gate delays directly affect the propagation times of signals through the layers of logic in a processor, which, in turn, determines the maximum operating frequency of the processor. Therefore, when lowering the operating voltage of a processor, one also limits its maximum reliable operating frequency. Over some useful range, which varies with processor, the maximum operating frequency will scale roughly linearly with voltage [90]. By employing frequency and voltage scaling, the system can reduce energy costs significantly by trading off total computational throughput.

With software-controlled clock and voltage regulator circuits, the system can permit *Dynamic Voltage Scaling* (DVS) [61], where the operating system sets the frequency and voltage of the processor to meet changing task needs. The basic idea behind DVS algorithms is to set the processor to the slowest speed necessary to complete all tasks, eliminating idle time, and then use the lowest voltage setting that permits this speed. The energy per cycle / operation will be dramatically reduced ($\propto V^2$). Assuming a nearly linear relationship between voltage and frequency, the combined effects of voltage and frequency scaling will reduce power dissipation nearly $\propto V^3$ (Figure 1.1(d,e)).

Once again, the main limitation in implementing DVS techniques in embedded and mobile platforms is that changing the processor frequency will affect task timings and disrupt any timeliness guarantees provided to the running applications. In this dissertation, *real-time DVS* (RT-DVS) algorithms are developed that can simultaneously reduce voltage to conserve energy and maintain deadline guarantees in a real-time system environment. The philosophy behind DVS – execute tasks as slowly as possible – runs counter to that of sprint-and-halt used with software-controlled power-down hardware, so completely different approaches and algorithms are developed to ensure real-time performance while exploiting these hardware energy-conserving mechanisms.

1.4 Energy-Aware Adaptation of Workload

The two broad approaches outlined so far ultimately improve the system's energy efficiency and allow a greater total amount of useful computation to be performed for a given amount of energy. However, there is no differentiation among the tasks in the system. In particular, these techniques cannot allocate limited energy resources to the most valuable or useful tasks. Furthermore, the DVS and sprint-and-halt mechanisms work best when there is a surplus of computing capacity. When there is no extra computing capacity, there is little room for reducing energy expenditure, which can be a serious concern if batteries are nearly depleted. In such a situation, something must be sacrificed — either the energy-conserving mechanisms are activated anyway, reducing computational throughput and causing all tasks to miss execution deadlines, or system runtime is forfeited.

The third major software-centric approach to energy conservation addresses this issue by adapting the system workload to better utilize the available energy. Here, mechanisms address how to best use and allocate limited stored energy among the computational tasks in the system. These will try to vary the service quality (in terms of processing capacity and energy) provided to each task in order to degrade system performance in a controlled manner, and to maximize the total value of computation performed with the limited energy budget. To implement this, the concept of Energy-aware Quality-of-Service (EQoS) adaptation of real-time workloads is introduced in this dissertation. The EQoS framework allows the system to automatically adjust the workload by varying service quality levels of tasks to meet system runtime goals. The service changes are applied independently to the different system tasks in order to provide better service and allocate more energy to those tasks that provide greatest returns or value from this energy consumption.

The adaptation of the workload to the available energy by varying service quality is a very open-ended problem. The EQoS framework seeks to formulate the problem and restrict it such that the adaptation is a tractable problem that can be efficiently solved. The problem is limited to the class of *known time-to-charge* problems. Here, one is given or estimates the amount of time the system must run on batteries before recharge occurs or primary sources of energy become available. The limited stored energy is allocated to the tasks such that:

1. the system runs for at least the time until recharge,
2. energy and computational resources are assigned independently to each of the tasks,
3. real-time performance is maintained or degraded in a known, predictable manner,
4. the total value of the computations performed by the tasks over the duration until recharge is maximized.

The EQoS framework requires each task to be assigned a simple scalar metric indicating the value or utility of the computational task. This abstract notion can cover any notion of importance, economic value, or user preference, for each task in the system. Furthermore, the degraded levels of service for each task are enumerated, and also assigned utility values. As embedded and real-time systems tend to have well-specified tasks, enumerating the degraded service levels does not present a large burden. Given energy estimates for each task, EQoS algorithms will select the appropriate service level for each task to achieve the needed runtime and maximize total utility over this runtime. The real-time requirements for the tasks at the selected degraded service level will be guaranteed, ensuring predictable, graceful degradation of the system.

The EQoS framework does not specify any particular mechanisms of executing real-time tasks at decreased service and resource levels. As this is a very application-specific issue, EQoS is meant to work with any RT degradation mechanism, and supports varying all real-time parameters. A control task may be executed at reduced service level by decreasing the frequency of execution (e.g., lowering sampling rate), while a audio task may switch to a low quality, low overhead CODEC (e.g., shorter average and worst-case execution times). For noncritical tasks, one possible degradation mechanism may be to simply not execute the task at all. EQoS adaptation algorithms need to handle all of these possible mechanisms of task execution with reduced service quality and still maintain schedulability of the real-time system.

In order to select the set of per task-service levels optimally, the algorithms need to be provided with some notion of the value or benefits gained from executing a particular task at various QoS levels. In addition to a quantification of task value / utility, the energy consumption of the tasks at reduced QoS levels needs to be provided. Energy measurement techniques for determining task energy requirements are developed in this dissertation. Using the characteristics of the tasks, available stored energy, and required system runtime, the algorithms select task service levels to maximize total system utility over the required runtime. One major hurdle in the design of these algorithms is that the actual task energy consumption is affected by RT-DVS and other mechanisms that conserve energy. In particular, as workload is reduced, RT-DVS reduces the processor operating frequency and voltage, improving its efficiency and reducing per-task energy requirements. Adaptation algorithms will need to consider this non-linear effect on energy consumption in order to truly optimize system utility for a given energy budget.

The adaptation of workload to energy does not improve the energy efficiency of a system. Rather, it seeks to automatically allocate limited energy to the various tasks in order maximize the value of running the tasks over a desired runtime and given energy budget.

Additionally, it will enable efficiency-improving mechanisms, such as RT-DVS, to work more effectively by scaling back the total workload when deemed necessary to meet run-time requirements.

1.5 Thesis Overview and Organization

This dissertation focuses on techniques to make operating systems energy-aware and improve energy consumption in embedded and mobile devices. Software-centric energy conservation is subdivided into three major approaches: improving service efficiency, controlling and exploiting hardware mechanisms, and energy-aware task adaptation. Although all three approaches receive some treatment here, software techniques of exploiting hardware power-reduction mechansims and energy-aware task adaptation are the primary foci of this work.

The following chapter considers methods of improving OS service efficiency to improve both performance and energy consumption. A part of this chapter is a survey of techniques proposed by the author and others in the context of the EMERALDS real-time microkernel, but some new work in regards to energy-conservation with reduced protocol processing for small Internet multimedia devices is also presented.

The third chapter presents methods of maximizing the benefits of software-controlled power-down hardware in a real-time system. In particular, *sprint-and-halt* algorithms [66] for scheduling power-down epochs are developed, with an emphasis on maintaining real-time execution guarantees. The algorithms introduced here can aggressively power-down a real-time system even when time overheads of switching power states are large.

Chapter 4 also investigates software control of hardware energy conservation mechanisms. Here, DVS techniques are considered. The impacts of these on real-time scheduled tasks are discussed, and a class of real-time DVS (RT-DVS) algorithms is introduced. Several novel RT-DVS algorithms are developed that ensure timeliness guarantees to real-time tasks while simultaneously exploiting voltage and frequency scaling hardware to greatly reduce energy consumption. In addition, an actual implementation of RT-DVS on top of Linux is presented and evaluated.

The third main approach to improving energy consumption is explored in Chapter 5.

Here, the EQoS framework for the energy-aware adaptation of quality-of-service provided to real-time tasks is presented. Key benefits of this framework are that it formulates and restricts the adaptation problem into a tractable problem, automates selecting between extra runtime and greater value through higher quality execution, introduces a clever method of accounting for improved efficiencies due to DVS techniques, and permits per-task adaptation while maintaining real-time service guarantees. A Linux-based implementation is also presented.

The EQoS framework is intended to work for embedded systems with tasks that are well-specified in terms of energy consumption. However, obtaining task energy characteristics may not be a trivial task. Chapter 6 explores methods of measuring energy at a time granularity corresponding to task execution, and develops a low-cost hardware measurement device that can provide accurate energy measures without requiring intrusive system modifications or complex statistical analysis. If even this is considered too costly, some simple models of predicting task energy requirements are investigated and evaluated against actual measurements.

Each of the technical chapters includes a discussion of related work, covering relevant research on OS optimizations, DVS techniques, and task adaptation methods. The final chapter presents some concluding remarks and comments on some potential future directions.

CHAPTER 2

Improving Energy Efficiency of OS Services

Improving software performance has always been a goal of software developers. Operating system performance is in particular well studied, as reducing OS overheads improves performance for all applications that use OS services. OS developers already use handcrafted code in critical sections and employ profiling and complier tricks to reduce service overheads and improve performance. From an energy consumption perspective, the reduced overheads translate into fewer processing cycles spent on a particular service routine, which directly translates into less energy consumed for the service. The saved energy and computing cycles are available to applications to perform useful work, reducing the energy cost per unit of useful work. Therefore, by optimizing an OS for performance, one also gains in terms of energy efficiency.

There are two philosophically-opposed approaches to improving OS service energy efficiency. The first approach maintains all service semantics and interfaces unmodified, ensuring all existing software work correctly as initially intended. With this approach, any improvements in energy and performance come from well-studied techniques such as compiler tricks (peephole optimizers, loop unrolling, etc.) and hand crafted inner loops or critical sections. These techniques are already used extensively in general operating systems, and there is limited scope for significant additional improvements.

The opposing approach is to identify characteristics of the type of platform and expected applications, and modify the standard service interface to best suit this expected use. By changing the behavior of the service, the system can be better optimized to the expected common-case scenario, rather than be saddled with a generic, general-purpose implementation. In particular, for handheld, embedded platforms, characteristics such as a generally static set of applications, or real-time scheduling, can be exploited to simplify or reduce the execution overheads of various OS services.

This chapter first surveys some of the techniques developed by the author and others for the EMERALDS operating system [95]. EMERALDS is a small, real-time microkernel intended for embedded systems. In the development of EMERALDS, OS services and interfaces have been modified to optimize for the expected use in small, embedded systems. These modifications can boost both performance and energy-efficiency of the services.

The second half of this chapter explores protocol processing and layer bypassing for Internet-enabled multimedia devices. Although originally proposed for EMERALDS [94], early work was based on archaic hardware assumptions. This chapter revisits protocol-layer bypass techniques under FreeBSD using modern network interface adapters, and evaluates the potential for energy savings.

2.1 Service Optimizations in EMERALDS

The EMERALDS (Extensible Microkernel for Embedded, ReAL-time Distributed Systems) is a small research operating system that has been under development at the University of Michigan. The goals of this OS are to provide an OS designed particularly for small-memory, embedded systems, with only the components necessary for such systems. Services in EMERALDS include hard real-time scheduling, synchronization and priority-inversion control, multithreading, and interprocess communications. Networking support (CAN control bus and IP protocol stacks) can be optionally included with the core OS services. As large non-volatile storage is not typical in most embedded systems, filesystem support is not included in EMERALDS.

The optimizations incorporated into EMERALDS focus on exploiting unique characteristics of embedded software on small platforms to modify OS service semantics to reduce processing overheads. By modifying service interfaces and behaviors, the system is optimized for the common-case scenario expected with embedded systems, reducing overheads and processing energy costs when compared to the unmodified, general-case service semantics.

2.1.1 Exploiting Static Characteristics

In most embedded and handheld devices, the actual application set is either static, or changes very rarely. Often, the entire set of application tasks is known and fixed when the system is built. In such systems, the shared resources among the tasks can be analyzed offline, and resource locators can be statically compiled into the tasks or inserted at a final link / load stage when the system is burned to ROM. EMERALDS uses this approach to eliminate naming and resource discovery services for application task using shared resources, such as shared memory segments, semaphores, etc. The overheads of accessing these resources is reduced by eliminating one layer of indirection, since the fixed locators are linked into the tasks.

Indirection is also reduced in OS system calls. The original EMERALDS system call mechanism uses direct addressing of the kernel service routines, with only an additional overhead of switching to protected kernel mode. This eliminates service call lookups and redirection, as well as the need for a generic interface to handle an arbitrary number of parameters. This requires that the tasks be compiled with, or finally linked against the symbol table of the kernel so the correct addresses of functions are used. Furthermore, when using virtual memory, the kernel address space must be mapped into each task's page table. As embedded systems tend to have small physical memory compared to very large virtual address spaces, this presents no real restrictions on applications.

Both of these mechanisms also work when tasks are dynamic. In this case, the final task loader/linker needs to perform the additional step of fixing the resource locator values or kernel function addresses into the task binary image. These mechanisms trade off increased complexity at system generation or application task load time, for reduced service overheads. As the task set is static or changes very infrequently in embedded systems, this one time or infrequently incurred additional cost is outweighed by the overhead reduction in every single system call performed.

2.1.2 Improved Real-Time Scheduling

Priority-based real-time schedulers assign priority levels to tasks, executing the highest priority task that is ready to run at any given time. The priority assignments may be fixed or dynamic, depending on the scheduling algorithm used. Fixed-priority (FP) scheduling is generally simpler, but, due to limitations inherent to the approach, may not be able to schedule task sets that fully utilize the available computational capacity. Depending on the frequency and execution times of the tasks, up to around 22% of processing capacity may need to be left idle to ensure all deadlines are met when using an optimal fixed priority scheduler [45].

With dynamic-priority (DP) scheduling, particularly *earliest-deadline-first* (EDF) scheduling, task priorities are changed continuously based on the current deadlines for the running tasks in the system. Ideally, this paradigm can schedule task sets that fully utilize the processor, assuming scheduling and preemption costs are negligible. However, the scheduling here incurs greater overheads than with an FP scheduler, since this requires dynamically sorting tasks by priority or scanning for the highest priority task at each scheduling point. Particularly in a system with a slow processor, or a large number of high-frequency periodic tasks, this scheduling overhead can be substantial.

To alleviate this, EMERALDS introduces the *combined-static-dynamic* (CSD) scheduler. With CSD, a carefully selected subset of the tasks are scheduled with DP techniques, while the rest are assigned fixed priorities. Fewer tasks in the DP subset means that the DP scheduling overheads are incurred less frequently, and as the dynamic task queues are shorter, each occurrence is reduced when compared to a purely DP scheduling approach. The remaining tasks incur only the lower FP scheduling overheads. The net effect is that CSD can schedule any task set that a DP scheduler can, but will incur lower (or, in the worst case, no greater) total scheduling overheads. On a 25 MHz 68040 platform with 10–20 real-time tasks, CSD techniques can reduce total scheduling overheads by 50–66%, recovering up to 3–5% of total processing capacity and energy consumption for use in application tasks [95]. Although the total processing capacity recovered will be smaller with faster processors, the relative improvement in scheduling overheads will remain.

2.1.3 Synchronization Issues

A very critical service in all operating systems is a mechanism of synchronizing access to shared resources by parallelly executing tasks in in order to prevent data corruption. Generally this will involve a semaphore or a simple mutual exclusion lock (mutex). When there is no immediate contention, the mechanisms are fairly efficient. However, when one task tries to access a resource that is already being used, the system will suffer significant overheads. In particular, a very common concurrent access situation is as follows:

1. System switches to Task A (starts / resumes from blocking call)
2. Task A tries to lock a mutex held by lower-priority Task B
3. Switch to Task B, due to priority inheritence [76]
4. Task B relinquishes lock
5. Switch back to Task A

This sequence entails 3 very costly context switches. EMERALDS addresses this by noting that if it is known at the start that Task A is going to access a lock held by Task B, the OS should directly switch to Task B until the lock is relinquished, and then switch to Task A, eliminating one context switch. This is implemented by modifying the interface to blocking system calls to provide a hint to the OS indicating the lock, if any, the application will try to access on resumption. As context switches dominate the service time, the new mechanism may eliminate up to 33% of the energy and processing overheads associated with concurrent access attempts to shared resources. Other work has also tried to reduce the overhead of semaphore operations by either relaxing the semaphore semantics [84] or devising vastly different synchronization policies [86].

One other method of reducing synchronization overheads is to try to eliminate the synchronization altogether. Interprocess-communication (IPC) requires either a heavy-weight mailbox-type approach, or a shared data area protected by a mutex. Various research projects [9, 38, 63] have devised techniques of IPC that involve no synchronization of data access, particularly for situations where there is a single writer and multiple readers. This is very common in embedded systems, such as in a task that samples and processes a sensor input, and publishes this information to all other tasks. However, such synchronization-free IPC mechanisms generally incur substantial processing overheads of their own. State messages and non-blocking write [34, 95] exploit the periodic real-time nature of many embedded systems, using the guaranteed execution timings and a set of cyclical buffers to ensure uncorrupted access, while reducing overheads to that of accessing unprotected globally shared data. Unfortunately, this may require a very large number of buffers, or the need to fall back to synchronization to keep memory use in check. More recent work, also incorporated into EMERALDS, extends existing wait-free IPC mechanisms to use temporal isolation [25], reducing processing and energy requirements 17–66%, while simultaneously reducing memory requirements.

2.1.4 Layer-Bypassing in Internet Protocols

In order to minimize energy expenditure in embedded network communication, one can devise custom energy-conserving communication protocols that incur very low processing overheads and minimize energy expenditures on handshaking, retransmissions, or other protocol control transmissions. However, embedded systems often need to communicate with external computers through standard wide-area networks. In such a case, support is needed for standard Internet protocols (TCP & UDP/IP). Unfortunately, these protocols were not developed with low-power, small embedded systems in mind, so they tend to be fairly processing- and energy- intensive.

In particular, significant processing overheads occur in the protocol stacks checking for error conditions. Extensive error checking is not needed in many embedded applications. In some applications, such as in video streaming to a handheld device, or audio-video conferencing, very little protocol-level error detection is needed, since any errors will only cause loss of some quality or transient "static" in the output. As these applications are soft real-time in nature, TCP's error-correction mechanism through retransmission is likely to provide stale data that must be discarded. Furthermore, some encoding techniques, particularly for audio, incorporate forward error correction (FEC) that allows the application to fix corrupted data. In such a case, quality of the output may actually be hurt by the error detection and drop at the protocol layers.

In this context, EMERALDS provides the option to bypass network protocol processing and, instead, allow application-level processing suited to the needs of the specific task. This is, in essence, a form of lazy-receiver processing (LRP) [14]. The novelty in the EMERALDS implementation comes from combining LRP with a single-copy architecture that performs any protocol processing within the buffers on the network interface, and copies data directly to user space, eliminating the intermediate copy to kernel memory before processing [96]. The single-copy mechanism does not require specialized hardware on the network interface as in some other approaches [12, 55]. The elimination of the copy step, applicable to all incoming packets, and the elimination / bypassing of general-purpose protocol processing can significantly reduce networking energy costs.

Unfortunately, the EMERALDS implementation is tied to an archaic network interface architecture that has multiple on-board receive buffers, and depends on the CPU to transfer packets to main memory. The rest of this chapter revisits layer-bypassing of Internet protocols and extends the work to support zero-copy techniques with a modern network interface architecture and streamlined network stack.

2.2 Protocol Processing in Embedded Internet Devices

Internet-enabled devices are becoming increasingly common and are evolving to handle a wide array of multimedia content available through on-line services, particularly through multimedia streams. In portable Internet devices, based on cellular phones or PDAs with wireless networking, energy consumption and battery life are major issues, so these devices will greatly benefit from any improvements to the energy overheads of Internet protocols.

In the design of an energy-efficient embedded system, one can readily devise a communication protocol that is very efficient in terms of computation overheads and energy consumption for handshaking, retransmissions, and other control-related protocol transmissions. Unfortunately, in the domain of handheld and mobile Internet devices, these small, embedded systems must be able to communicate with computers world-wide using standard Internet protocols (TCP and UDP). These protocols were designed primarily with the goals of decentralized control, robust communications, and an end-to-end paradigm, and were not designed with low-power embedded systems in mind. As a result, TCP and UDP require substantial resources at the end hosts, and tend to be fairly processing- and energy-intensive.

However, in the domain of multimedia on hand-held devices, there are some optimizations that can be made to reduce the computation and energy loads incurred with IP. In particular, a significant part of the network protocol processing is spent on two main activities: testing for error conditions, and computing checksums. The error checking detects relatively rare events, such as lost packets or out-of-order packets, as well as handling easily-avoided issues such as fragmentation and reassembly. These services are needed in general-purpose Internet protocols to ensure robustness on a wide variety of underlying network technologies and to adhere to the end-to-end ideal. For multimedia on hand-held devices, particularly for streaming audio and video, however, much of this will not be used. Fragmentation can be avoided by performing a simple Maximum Transmission Unit (MTU) discovery protocol prior to streaming the data to ensure packets are not broken up by the network. Since audio and video are soft real-time applications, late data is not useful past a certain time, dictated by the frame rate or sampling interval. Therefore, retransmission of dropped or corrupted packets is not useful for these applications unless a fairly long duration of data is buffered ahead of time. The use of forward error correction in streamed multimedia can better recover from corruption without incurring any delays due to retransmissions, and can be negatively impacted by protocol-level error detection through checksumming. Furthermore, streamed audio and video are meant to be viewed by human users, so unlike other kinds of data communication, they are fairly tolerant of errors. Any lost or corrupted packets of data are not catastrophic, and merely result in degraded quality, often perceived as static or transient skips. The computationally-intensive, and therefore energy-intensive, task of error detection through checksum computation in the UDP and TCP protocols is unnecessary in these applications.

As a result, most of the Internet Protocol stack processing for received multimedia streams may be entirely bypassed, saving considerable computation time and energy. A modified networking stack on Internet multimedia stream player devices can employ a packet filter [19, 54] in the lowest software device driver level to quickly sort the multimedia packets from other data packets. While the latter will traverse the ordinary protocol stacks to benefit from the robust, end-to-end communications provided by the Internet protocols, the majority of the packets (audio and video) will be diverted to an energy-conserving alternate stack. This will bypass the costly checksumming and error correction steps, or deliver the packets directly to the application, allowing for application-level protocol processing or LRP[14]. By combining this with a single-copy architecture, as implemented in EMERALDS, significant energy and processing overhead reductions can be realized.

Unfortunately, the EMERALDS implementation is strongly tied to a particular network interface architecture not commonly used today. Therefore, we extend this work to implement protocol layer-bypassing and zero-copy mechanisms using modern network interface card (NIC) architectures. In particular, UDP packet reception on a FreeBSD kernel is modified to to reduce energy and processing overheads in streamed multimedia applications.

2.2.1 Network Interface Architectures

The EMERALDS layer-bypassing and single-copy mechanism provides low-overhead networking for embedded applications, without requiring significant support or specialized features on the NIC hardware. It assumes a NIC architecture similar to the LANCE chipset, an unassuming architecture that provides a basic Ethernet interface. The NIC uses on-board buffers to store received packets, and depends on the processor to transfer them to main memory, typically a kernel buffer where protocol processing occurs. After this, data is copied a second time to the user task's buffers. The novelty in the EMERALDS approach is that the kernel preforms any processing directly in the NIC receive buffers, requiring only a single data copy to the application buffer.

Most current NICs do not have this type of architecture. In particular, they do not use on-board buffers that are exposed to the host processor. Instead, to alleviate processing overheads, they employ direct memory access (DMA) or busmastering techniques to directly write received packets into kernel buffers. Hence, they are not amenable to the EMERALDS approach. How layer-bypassing may work in a system that uses a DMA-based NIC is considered next.

2.2.2 Packet receive operation

Normal operation of a datagram packet receive is shown in Figure 2.1. The application first sets up a socket to accept packets on some UDP port. It then creates a buffer to receive the packet and makes a receive call that causes it to block until a packet is available. On the system side, nothing occurs until a packet arrives at the Ethernet interface. The card uses DMA to write the incoming packet to a preassigned kernel buffer, called an *mbuf* [91]. All of this occurs entirely in hardware. Once the packet has been buffered, the hardware signals an interrupt to the processor, which then jumps to the device driver interrupt handler in the

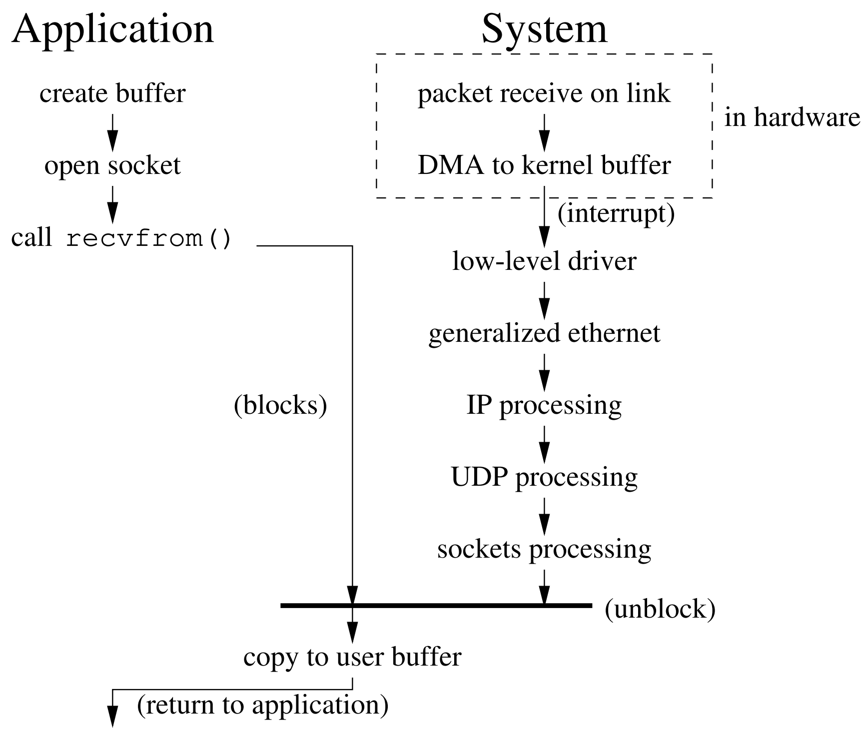


Figure 2.1: Normal datagram reception

kernel. After a quick check to see if a packet arrived (since the card may signal interrupts on events other than packet arrival) and some hardware state clean-up code (which includes assigning a free mbuf for a future packet), the low-level device driver passes the packet up to the generalized Ethernet driver, the lowest software level of the protocol stack. The packet is processed and passed up through the IP and UDP layers to the application interface (sockets) level.

A few optimizations are already incorporated in BSD to reduce the overheads of protocol processing. The packet is "passed" between protocol layers by simply passing the pointer to its mbuf, and as much as possible, the packet is processed *in situ*, thus avoiding costly memory-to-memory copying. As a further optimization, the interrupt handler relinquishes control of the processor in the generalized Ethernet level, limiting the duration of preemption due to the hardware interrupt. A separate kernel routine, triggered as a "software interrupt," continues packet processing for the higher protocol layers.

When processing is complete, the receiving task unblocks and the receive system call continues. The system call copies the packet into the application's buffer, checking for buffer length, segmentation problems, and other errors that may occur with application-specified pointers. Finally, the receive call completes, returning control to the application.

With effective use of *in-situ* processing, this receive operation essentially implements a single-copy architecture, since the processor needs to perform only the final memory-to-memory copy into the application receive buffer. However, this still entails two major data-touching operations: performing the UDP checksum and copying data to the task buffer. Layer-bypassing will avoid the former, while a zero-copy extension will eliminate the latter's overheads.

2.2.3 Modified packet receive operation

This work implements a version of IP layer bypassing on the FreeBSD 2.2.8 kernel. The approach taken involves bypassing the entire receive-side protocol stack above the device driver, and directly delivering received packets to the applications. To do this, a new system call is introduced that applications can use to request layer-bypassing and retrieve received packets. This call provides semantics similar to the standard receive from socket call for UDP, but internally sets up packet filtering and performs layer-bypassing to avoid the computational and energy overheads of UDP/IP.

In order to evaluate the overhead reductions of the layer-bypass mechanism, protocol timing support is also implemented in the kernel as well. A very low-overhead mechanism (just a few machine instructions long) records a timestamp for any packet that contains a "magic number" as it is received or sent out at the lowest software driver level. This uses a 64-bit hardware cycle counter found on all Pentium-class and higher Intel microprocessors, and can easily provide sub-microsecond precision. A second system call is introduced that returns this timestamp to the benchmarking application to determine total protocol processing times. Figure 2.2 summarizes the API for these two new system calls.

The UDP packet recieve operation in the modified kernel is outlined in Figure 2.3. From the view of the application, everything looks the same, except that the new `direct_deliver()` call is used. In contrast, on the system side, differences occur as soon as the interrupt handler begins execution. After the usual checks and state clean-up, the device driver performs some minimal packet filtering, quickly scanning for particular packets with little incurred overhead. In this case, packets are checked to see if they contain a particular "magic number" in the header; if so, these are assumed to be packets used for timing

```
int direct_deliver ( int sockfd, char* buf, int buflen )
```
Receive packet on UDP socket `sockfd`, return it in buffer `buf` of length `buflen`.

```
int proto_time ( int svc )
```
Retrieve receive time (`svc` = 1) or transmit time (`svc` = 0) at the device level for the most recent protocol timing packet (contains magic number). Returns the lower 32 bits of the CPU cycle counter at time of packet arrival.

Figure 2.2: API for new system calls

the protocol stack, and the arrival time (64-bit processor cycle counter) is saved. This can be done very quickly, since the test is a simple read and compare, and the timestamp is obtained through a single machine instruction. A second test compares the destination port of UDP packets with the receiving application's port; if it matches, then layer-bypassing occurs, the packet mbuf is added to a receive queue, and the task is unblocked before the interrupt handler exits. The fail-through path of this filtering step follows the normal protocol processing steps. Once the task has unblocked, the system call will copy the packet into the application's buffer and return control as in the normal processing case.

This implementation ensures that certain aspects remain completely compatible with unmodified code. In particular, the application socket setup mechanism is identical to the normal networking case, so the only apparent difference to application is the system call used to retrieve the packet. The new system call takes care of any state manipulation setting up and removing bypassing transparently. On the other hand, in the low-level packet reception code, elimination of overhead is the goal. Instead of extending the general-purpose Berkeley Packet Filter (BPF) to support the layer bypass, a very small, custom filter is used in this design, sacrificing widespread applicability for smaller, faster code.

2.2.4 Layer-bypass Performance

Several experiments have been performed to test the improved efficiency obtained from this layer-bypass mechanism. These experiments are run on the FreeBSD testbed illustrated in Figure 2.4. It consists of four HP Kayak XA computers, with 450 MHz Intel Pentium II processors, connected by switched 100 Mbps Ethernet. Since the layer-bypassing begins at the lowest software level and requires modifications to the network interface driver, all of the machines use the same type of Ethernet card (SMC EtherPower II, 100 Mbps). All

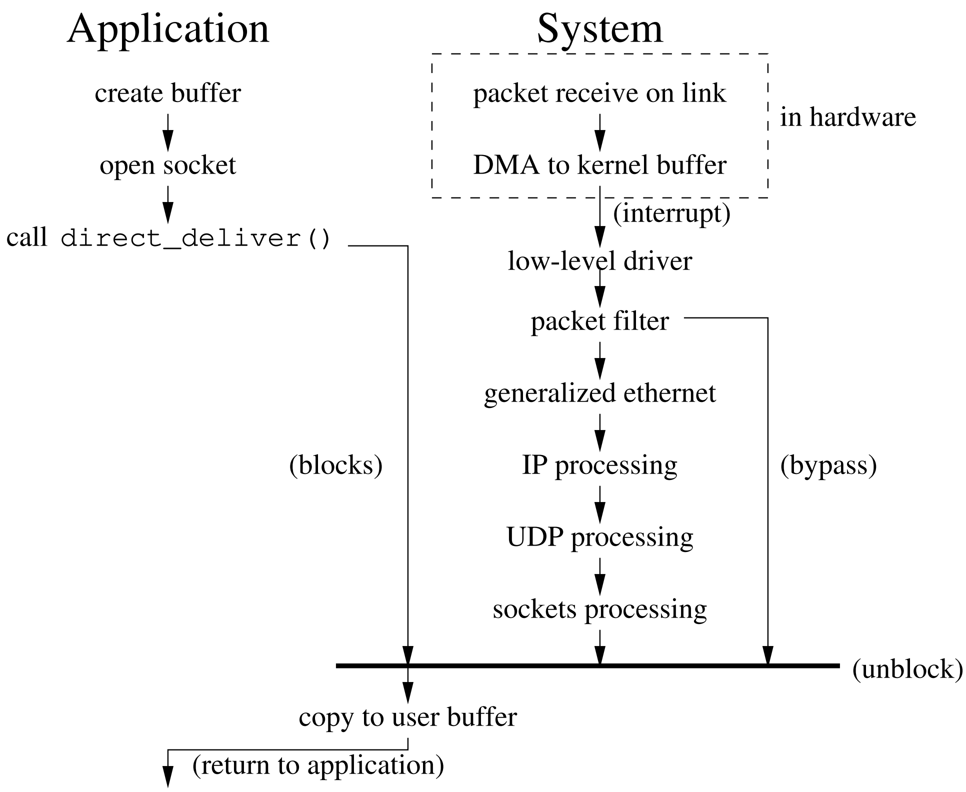


Figure 2.3: Normal datagram reception

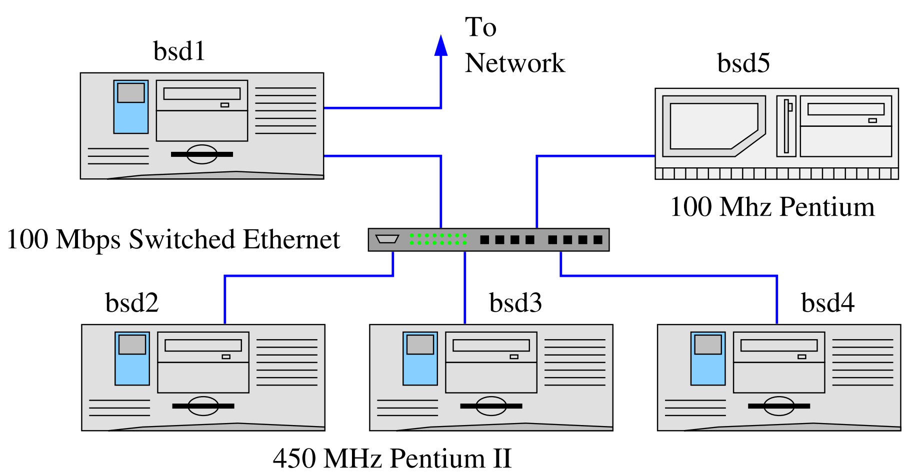


Figure 2.4: Testbed

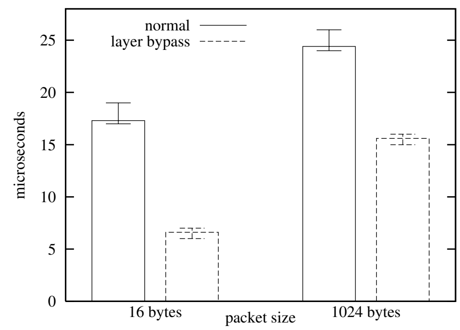


Figure 2.5: Receive-side processing latency

are running FreeBSD 2.2.8, with modifications introduced on machines *bsd2* and *bsd3*. A fifth computer, *bsd5*, also running the modified kernel, is an older generation platform and has a Pentium 100 Mhz processor. This last machine is intended to be much closer in performance to Internet appliances and representative of the processing power to be available on hand-held and embedded devices today, while the former are more representative of the embedded, mobile processing power expected in the near future.

UDP Receive Processing Overheads The first set of experiments compares total packet receive protocol processing overheads with and without the normal layer-bypassing mechanism. Machine *bsd2* runs an UDP echo server application that simply receives any packet on its listening port and sends it back to the source. *Bsd3* performs the actual protocol processing time measurements. The measurement application creates a datagram packet containing the magic number to ensure that the kernel records the transmit and receive times at the device driver. The application records the transmit and receive times seen at the user level just prior to the send system call and just after returning from the receive system call. The difference between the device driver and application level numbers indicates the duration that the packet spends in the kernel for protocol processing, data copying, and other related kernel overheads.

The total latency between packet arrival at the low-level device driver code and receive completion at the application level is measured for both small (16-byte UDP) and large packets (1024 byte UDP). Figure 2.5 shows the average times measured and indicates the range of values observed. Overall, the layer-bypass mechanism is able to significantly reduce the processing overheads, cutting processing between 36% and 62%, depending on packet length. Interestingly, the longer packet actually exhibits a lower percentage savings than the shorter one. One would expect that due to fixed overheads (such as from blocking or system call mechanism), there would be diminishing returns for the small packet size and that the improvement would be lower than for the long packet.

This counter-intuitive behavior can be explained by looking closely at the processing steps in Figures 2.1 and 2.3. There are normally two main data touching steps: the UDP processing (due to checksums) and the final copying to application memory. These operations are not independent. Specifically, by performing the UDP checksum, the processor reads the entire packet into the data cache. As a result, the memory copying is greatly accelerated, since data reads are served from the cache rather than from the relatively slow main memory. The layer-bypassing mechanism eliminates the extra processing of the checksum, but also eliminates most of the beneficial prefetching to cache. As a small, fixed part of the packet is cached anyway as a side effect of the header examination, the memory copying for short packets suffers less than for the long ones, resulting in better percentage gain for short packets with layer-bypassing.

Elimination of blocking effects

The total receive side latency includes protocol processing, task blocking, and memory copying overheads. To determine how much of this is due to blocking and context switching overheads, a second series of experiments uses nonblocking receive mechanisms to eliminate blocking from the measured latency. In general, this does not involve changes to the kernel, as support for nonblocking socket calls are already built-in. However, the new layer-bypassing receive mechanism does need minor modifications to support nonblocking calls.

The experimental setup is identical to those before, except that the measuring task now uses nonblocking calls. The task enters a polling spin-loop, continuously making the nonblocking receive call until it succeeds. Because of the nonblocking call, the task is never switched out, avoiding one context switch and the overheads of updating the kernel task structures on each receive, resulting in a shorter receive-side processing latency.

The results of these experiments are shown in Figure 2.6. Overall, blocking effects seem to incur a fixed overhead of approximately 2μs regardless of packet size or whether layer-

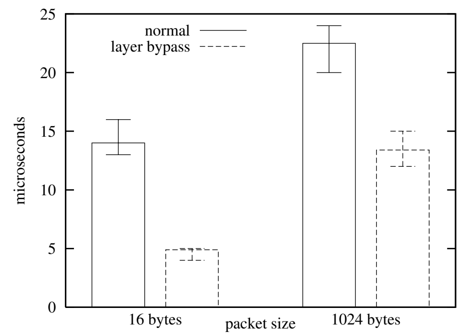


Figure 2.6: Receive-side processing latency, nonblocking calls

bypassing is used. As a result, the absolute latency reductions from layer-bypassing are not significantly changed. The percentage improvement is somewhat greater (40% to 65% depending on packet length) because of the smaller values used as the basis of comparison.

Although these numbers show a shorter latency, using nonblocking will not be very useful in the context of conserving energy. Any savings from the eliminated context switch and reduced kernel structures updates are overshadowed by the wasteful polling and spin-loop in the application that performs no additional useful work. However, in an application where latency is the most critical factor, this nonblocking, polling can shave an additional couple of microseconds from the receive latency.

Effects of slow processors

The experiments above were all performed on machines with processors that are faster than those in handheld devices today, but comparable in performance to high-performance handhelds expected in the near future. To evaluate benefits for platforms with lower-performance processors, as can be expected in today's hand-held multimedia devices and PDAs, these experiments are repeated using a computer with a 100 MHz Pentium processor. The latest PDAs, with 400 MHz XScale processors, may already be faster than this machine when peak processor throughput alone is considered.

In this series of experiments, the echo server continues to run on *bsd2*, but the timing task is now executed on *bsd5*, the slower computer. Nonblocking calls are still used here. The results are summarized in Figure 2.7. Clearly, the total processing time is much greater on the slower machine, but not the 4.5x that the processor frequencies may imply.

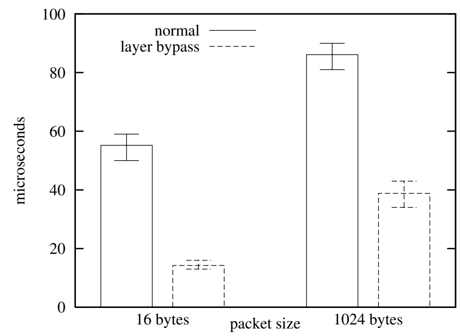


Figure 2.7: Receive-side processing latency with slow processor

This is because not all of the system scales up with processor speeds. In particular, the main memory on the faster machines is only clocked only 50% higher than on the slow machine (100 MHz vs. 66 MHz). Despite this, the relative savings due to layer-bypassing (55% to 74% depending on packet length) are greater than for the fast machines. This is a consequence of the improved architecture in the newer processors, which can perform the computations such as the UDP checksum in fewer cycles, so they do not benefit as greatly (in terms of latency) as the older-generation processor from layer-bypassing. Regardless, the total processing overhead reductions are very large in both the slow and fast processor configurations.

2.2.5 Zero-Copy Extension

In the previous section, layer bypassing is used to eliminate processing overheads of protocol processing for multimedia applications. In particular, data-touching operations for checksumming the received packet is removed. However, there still remains one expensive memory-copying step to transfer the data to user buffers. Here, a zero-copy extension is used to eliminate this overhead as well.

The approach taken in this extension uses the virtual memory system to remap buffers between the kernel and application memory spaces, avoiding the need to copy data for the final step of a packet receive operation. In particular, the new receive system calls are modified to swap the application-specified buffer's physical memory page with that of the kernel mbuf cluster containing the packet. Page remapping has been used to reduce copying in [11] and [88], but the particular physical page swapping implemented here requires much less modification of the VM system, as page allocation is unaffected and there is no net change in the number of pages held by either the kernel or the application task when remapping is performed. The approach taken in *fbufs*[15] also provides buffer remapping, with potentially lower overheads than the approach here, but it also requires extensive VM and OS modification to use fbufs rather than the existing buffer mechanisms.

In order to use the simple physical page-swapping mechanism to remap the kernel and application buffers, several key criteria must be enforced. Most importantly:

1. There must be no other data sharing the physical memory page with the buffers.

2. The buffers must fit completely within the page, and not span two or more memory pages.

As the page size on Intel processors is 4096 bytes, and the Ethernet MTU is 1500 bytes, buffer length is not an issue. Since the kernel always allocates new mbuf clusters aligned to page boundaries, the latter issue is not of concern. To fix the first issue, the mbuf cluster size is increased from the default 2048 bytes to 4096 bytes, thus ensuring no other data shares the page. This does increase memory usage, but since the number of outstanding mbufs is typically low (a few dozen), this does not present a very heavy burden. On the application side, the only sure approach to having a clean memory page for the buffer, without modifying the memory allocator, is to allocate twice the needed memory (8096 bytes) contiguously, and use a page aligned block within this as the buffer.

The receive calls now swap the application buffer and mbuf physical pages by changing the entries in the hardware page table. In addition, they must change a variety of data structures in the kernel to ensure that the view of memory page allocation remains consistent. Finally, the translation look-aside buffer (TLB) of the processor, which caches page table information to accelerate memory access, is flushed of the relevant entries.

All of this complexity is to provide the zero-copy approach on systems with memory protection. On an embedded platform that does not have or does not use memory protection, this becomes trivial. The application can access any memory, so the receive calls need to only return the pointer to the kernel buffer containing the received packet.

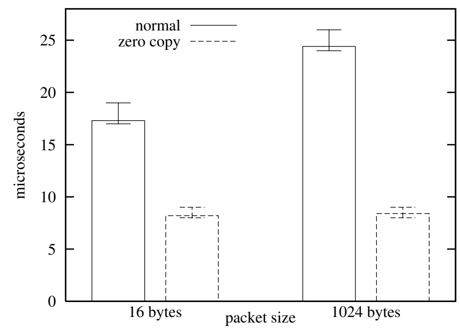


Figure 2.8: Receive-side processing latency, zero-copy

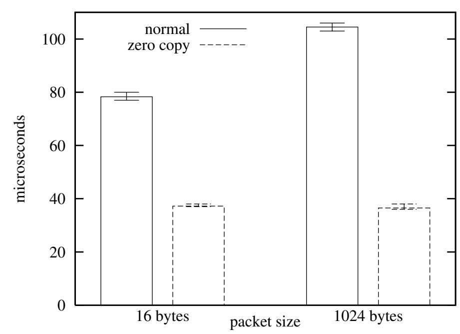


Figure 2.9: Receive-side processing latency, zero-copy, slow processor

2.2.6 Zero-copy Performance

The combined layer-bypassing and zero-copy mechanism is re-evaluated on the same testbed. In these experiments, once again, *bsd2* is the echo server, and *bsd3* or *bsd5* host the measurement application. Blocking calls are used in this series of experiments. The results of the combined layer-bypassing and zero-copy mechanisms are shown in Figures 2.8 and 2.9.

The first thing that stands out is that with the combined layer-bypass and zero-copy architecture, the receive side latency is constant for both short and long packets. This is a direct consequence of eliminating all data-touching overheads in software. The only data-touching step is in transferring the packet from the network interface to kernel memory, but this is all done by DMA in hardware, and does not incur processing overheads. As a result, the improvement is much greater for long packets than for short ones (approx. 52% to 65% on both the slow and fast processors).

Secondly, when compared to the results from layer-bypassing alone (Figure 2.5), latencies are increased for short packets when the zero-copy mechanism is introduced. This is a direct result of the overheads introduced by changing the page table entries. Although modifying the hardware page tables is straightforward and fast, the page structures in the kernel require wading through a variety of complex structures and lists. Moreover, it is necessary to flush the entries from the processor TLB, which causes further slow-down as the entries are fetched on subsequent memory access. These overheads are quite large in relation to the few cycles needed to copy a 16 byte packet. However, for long packets, a substantial savings is realized by avoiding the memory copying step.

2.3 Conclusions

One critical aspect of a software-centric approach to improving the energy efficiency of embedded and mobile devices is to improve the energy efficiency of the operating system services. This entails reducing the processing overheads and the corresponding energy consumption of the services. A survey of some of the techniques implemented in the EMERALDS operating system shows that reductions in service overheads can be achieved when service semantics are changed and optimized for the intended embedded applications. Most show modest, but measurable reductions in processing overheads.

The more in-depth study of protocol layer-bypassing and zero-copy buffering shows significant reductions in processing overheads can be achieved for multimedia streaming applications that do not need much of UDP/IP error handling mechanisms. Significant improvements of up to 65% of the OS service overheads in a packet receive operation may be eliminated for such applications. Based in the power consumption of the Intel Pentium and Pentium II processors used in the experiments [30, 31], up to 0.7mJ may be saved per packet received. Of course, if layer-bypassing is used to simply implement LRP techniques, then most of the protocol overheads will simply be deferred, and the saved energy and processing cycles consumed later.

The various techniques of reducing service overhead can impact the energy impact of an operating system, freeing processing and energy resources for more useful computations. When this surplus capacity is used by applications on useful work the energy efficiency of the system in terms of energy cost per unit of useful computation, is increased. However, the main drawback of relying solely on overhead reduction to improve energy-efficiency, is that if the system cannot make use of the extra time gained, energy will be squandered in idle loops. Hence, these techniques will work best in conjunction with other techniques, such as software-controlled hardware power-down, that can reclaim energy that would otherwise be wasted in an idle system.

CHAPTER 3

Sprint-and-Halt Scheduling for Energy Reduction in Real-Time Systems with Software Power-Down

The previous chapter introduced various techniques to optimize OS services for small, embedded systems and thereby improve the energy-efficiency of computational tasks. This chapter investigates a second aspect of software-centric techniques of improving embedded system energy efficiency — algorithms for maximally exploiting hardware energy-conserving in a real-time system context.

This chapter introduces a class of *sprint-and-halt* schedulers that attempts to maximize the energy savings of software-controlled power-down mechanisms, while simultaneously maintaining hard real-time deadline guarantees. Several different algorithms are proposed to reclaim unused processing time, defer processing, and extend power-down intervals while respecting task deadlines. Sprint-and-halt schedulers are shown to reduce energy consumption by 40–70% over typical operating parameters. For very large or small state transition latencies, simple approaches work very close to theoretical limits, but over a critical range of latencies, advanced schedulers show an additional 10–20% energy reduction over simpler methods.

3.1 Introduction

In recent years, there has been a significant shift toward mobile computation and communication platforms and devices. This shift has occurred in both the realm of general-purpose computing with an increase in the use of laptop computers and PDAs, and in the embedded computing realm with the increasing popularity of digital cameras, cellular phones, and portable medical devices running complex applications and operating systems on embedded microprocessors. Critically constraining these systems is the limited stored energy available in a portable form factor, as there is a fundamental trade-off between the weight and size of the device, the processing speed of the processor, and the useful battery life of device.

Unrelenting market pressures have created increasingly-sophisticated applications in increasingly-compact devices, such as multimedia and web capabilities on cell phones and gaming on PDAs. These demanding applications require the use of more powerful processors to provide the user a responsive experience. Attempting to use simpler, less-capable processors to improve battery-life in consumer devices is not a very marketable option. This has made the need for power management to minimize energy waste in such systems critical.

There has been recent interest and significant research on *Dynamic Voltage Scaling* (DVS) techniques [6, 23, 90] that attempt to trade off performance and battery life by adjusting the operating frequency and voltage of the processor to match the computational load on the system. As processors are composed mostly of CMOS logic gates, the energy expended is proportional to the charge on the gate capacitances, and thus, a quadratic improvement in energy is attained when voltage is reduced. However, DVS requires software adjustable voltage regulators and clock generators that may not be available on many platforms.

More generally available is the much simpler concept of a *software-controlled power-down* mechanism. This may take a variety of forms. One simple form is a processor `halt` instruction that will effectively stop the CPU core, and keep it in a low-power, standby state until a subsequent interrupt. This is a low-overhead, fast operation that can simply be invoked in place of an idle-loop to reduce wasted energy by the processor. More generally, there may be some mechanism to place various system components into a standby state, incurring a finite time overhead to power down and up the system.

This time overhead of switching hardware power states adds complexity to managing power in embedded devices. In particular, these devices often require strict timeliness guarantees for executing their resident tasks. In such systems, any adjustment of hardware power states must ensure task deadlines are not violated while maximizing energy savings.

In this chapter, we propose and evaluate a class of *sprint-and-halt* scheduling algorithms that provide real-time task scheduling, while maximizing the benefits of software-controlled power-down mechanisms. The rest of this chapter is organized as follows. We will first present our general model of software-controlled power-down hardware, which is followed by a detailed description of several algorithms for sprint-and-halt scheduling of real-time systems. We then evaluate these algorithms with respect to energy savings, before ending with conclusions and a discussion of future work.

3.2 Background

As power dissipation becomes an increasingly critical limitation in mobile systems, various mechanisms have been introduced to help conserve and reduce wasted energy. The most general type of power conservation mechanism is based on changing the power state of hardware components, placing them in a low-power or standby state when not actively used. For general-purpose systems, mechanisms such as APM and ACPI [1] provide interfaces for software-controlled power-down of the system when not actively used. When explicitly notified by the user, such as when closing the lid on a laptop computer, or after some timeout interval without user input, the system enters a low-power state, and computation is halted until a subsequent wake-up event occurs. This works well in laptops and PDAs, which are usually idle when a user is not directly interacting with the system. However, most real-time applications are not considered interactive, and generally need to run continuously over extended periods of time. Battery-operated embedded systems cannot take advantage of simple timeout-based power-down to conserve energy.

Instead, for such systems, we need to take advantage of power-down mechanisms at much finer time-scales, halting operations between executions of periodic tasks. One hardware power-down mechanism that works well here is a processor halt operation. Here, a special `halt` instruction puts the processor to a sleep mode, turning off the execution pipeline and disabling further computation. Although power is still supplied to the processor, along with a clock signal, much of the CPU core is deactivated and power dissipation is very low. A subsequent event, generally a hardware interrupt, will resume the processor core. Using the halt instruction in place of a more traditional idle-loop can greatly reduce the wasted energy executing empty spin loops. As the overhead of executing the halt operation and resuming on interrupt is very low, on the order of a few processor cycles, this mechanism may be safely employed without significantly affecting execution times or deadlines.

The halt instruction, if available on the processor, is effective for conserving energy, but only within the processing core. The rest of the system, including buses, memory, and communication devices, will continue to draw energy at normal rates even when the processor is halted. With more sophisticated hardware, a system can provide an interface that allows a larger subset of system components to be deactivated under the control of software. Timers, memory controllers, and communication ports may be powered down to save considerable energy when not in active use. In particular, turning off the main system clock-generation circuitry will essentially shut off the processor and memory, and often any communications and I/O subsystems as well, saving considerably more energy than with processor halts alone. Taking this to an extreme, APM and ACPI suspend modes essentially turn off the entire system after saving all dynamic processor and system state to persistent storage, dropping power dissipation to zero.

These lower-power modes do not come for free. Unlike the simple processor halt, powering down external subsystems can incur substantial time and processing overheads for entering and leaving the low-power modes. In the extreme case of the ACPI suspend operation, the operating system must iterate through every system device driver, saving the current state, and then copy all memory to disk before powering down the system. Upon resume, this process is reversed to restore the system to the exact state it was previously in. The overheads for such operations is very high, and will typically require on the order of tens of seconds to complete. Although this can greatly reduce power consumption, it cannot be used in the short intervals between task invocations in a real-time system. At the opposite extreme, the halt operation incurs negligible to very low overheads, on the order of a microsecond, but may not provide significant energy savings in a system whose energy consumption is dominated by components other than the processor. Trading off power reduction level for improved switching latencies, a moderately aggressive approach

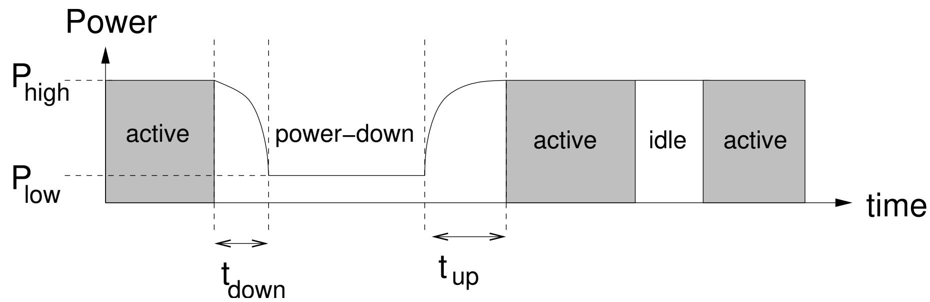


Figure 3.1: Parameters of system power model.

may require only a few milliseconds of overhead, such as in waiting for the clock circuitry to stabilize on power-up. Various other software-controlled power-down mechanisms can vary anywhere between these extremes, trading off power reduction for time overheads.

3.3 System Model

Regardless of the actual software-controlled power-down mechanism available in a particular hardware platform, its use in a real-time system is primarily affected by the time overheads the mechanism incurs and how this would affect the timeliness of task execution. Therefore, we can generalize software-controlled power-down mechanisms and model them as follows. First, for simplicity, we assume that the platform dissipates power in a bimodal manner, consuming a constant P_{high} when in the active state, and P_{low} when in the power-down state. The transition from active to power-down state takes a constant time, t_{down}. We assume that there exists some time trigger, such as an external real-time alarm, that can be programmed to reactivate the system at a specific future time. Once triggered, the transition to the active state takes t_{up} time. The average power dissipation during the transitions is P_{trans}, which can be anywhere between P_{low} and P_{high}, but we will assume the worst case of $P_{trans} = P_{high}$ unless noted otherwise. These parameters are illustrated in Figure 3.1.

We assume that the system follows the canonical periodic real-time task model. Each task i is *released* periodically, becoming ready to execute every t_i time units. The task is also characterized by a *worst-case execution time* (WCET) C_i, which indicates the maximum processing time it needs on each release/invocation. The relative deadline is equal to

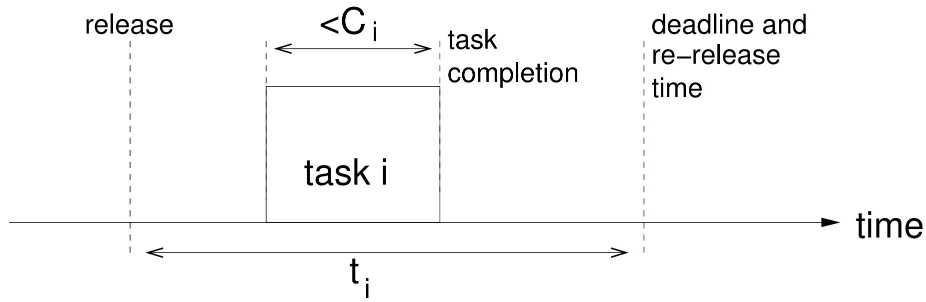


Figure 3.2: Periodic real-time task model parameters.

the period, so each task must complete execution within t_i of its release, i.e., must complete by the time it is re-released for its next invocation. These parameters are illustrated in Figure 3.2. The tasks are scheduled according to either the *rate-monotonic* (RM) or the *earliest deadline first* (EDF) priority scheduler. These are the most extensively-studied real-time scheduling mechanisms and cover a broad range of actual OS implementations. RM is a preemptive scheduler that assigns fixed priorities among tasks, giving the highest priority to the most-frequently executed task. EDF, on the other hand, assigns dynamic priority based on which task has the most imminent deadline, which varies over time. Assuming preemption and scheduler overheads to be negligible, the latter has a nice schedulability property that allows one to ensure a set of tasks is schedulable and all deadlines met by simply keeping the total worst-case processor utilization of the task set below one, i.e., $\sum C_i/t_i \leq 1$ [45].

Using the system models described above, we will in the next section design real-time scheduling algorithms that attempt to maximize energy savings from powering down the system, while ensuring real-time deadlines are met.

3.4 Sprint-and-Halt Algorithms

Existing real-time scheduling algorithms were not designed with energy-savings in mind. In particular, they do not consider how to incorporate software-controlled power-down mechanisms in the task schedule, and how to deal with the latencies incurred when switching between power states. In this section, we develop several novel algorithms to take advantage of power-down techniques while ensuring the schedulability of the real-time task set. These algorithms attempt to rapidly complete all work in the system (thus the term "sprint"), and then power down the system as long as possible (thus the term "halt") to maximize the reduction in energy consumption and amortizing the transition latencies over long power-down intervals.

3.4.1 Real-Time Schedulers with Power-down

We first consider the standard RM and EDF schedulers and extend them as minimally as possible to incorporate power-down control in the task schedule. The goal of this first design is to ensure schedulability and task deadlines by leaving the actual execution schedule unaltered. Rather, this algorithm incorporates power-down such that all execution timings are left identical to that of plain vanilla EDF or RM scheduling.

This algorithm tries to replace any idle time in the schedule with a power-down event while preserving task timing. However, due to the latency of power-state change, power-down must be applied only when idle periods in the schedule are sufficiently long to cover the transition latencies. Based on the model parameters specified earlier, power-down is triggered only when $t_{idle} \geq t_{down} + t_{up}$, where t_{idle} is the contiguous idle period in the schedule. When power-down is invoked, the system is set to resume execution in $t_{idle} - t_{up}$ time, ensuring that the system is in the active state by the time the idle period expires.

Given the real-time assumptions of a task's relative deadline equal to its period, and a work-conserving RM or EDF scheduler, one can very easily compute t_{idle} online. When some task completes execution and no other tasks have any computation time remaining, an idle period in the schedule begins. This idle ends upon release of the next task, which, since the relative deadlines equal the task periods, will coincide with the earliest deadline among the tasks within the system. Hence, $t_{idle} = D_1 - t_{now}$, where D_1 is the earliest deadline in the system, and t_{now} is the current time when idle would normally start. Figure 3.3 shows a pseudocode implementation of this algorithm. For EDF scheduling, the set of tasks is already sorted by deadlines, so adding power-down is trivial. For RM scheduling, one needs to add structures to keep track of the deadlines. In practice, it is not necessary to actually sort the task set by the deadlines, as a simple scan to find the earliest deadline is sufficient.

This algorithm is very conservative, avoiding altering any timing from the normal EDF

Assume n tasks, sorted by deadline:

$D_1 \leq D_2 \leq \cdots \leq D_n$

/* this is already needed for EDF */

upon task_release(task i):

set $done_i$ to 0;

update D_i to $D_i + t_i$;

resort task list by new deadlines;

/* schedule by RM/EDF priority */

upon task_completion(task i):

set $done_i$ to 1;

if (for all j, $done_j$=1) then:

t_{now} = get_current_time();

if ($(D_1 - t_{now}) > (t_{down} + t_{up})$) then:

set wakeup timer to $D_1 - t_{now} - t_{up}$;

start power-down;

else idle;

else:

/* schedule by RM/EDF priority */

Figure 3.3: Real-time scheduling with power-down

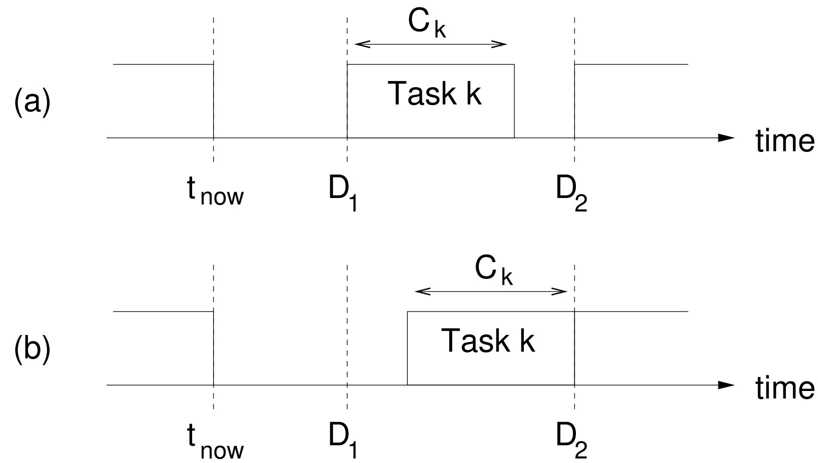


Figure 3.4: Example of deferral of task execution in work-idle-conserving scheduler. (a) Original execution schedule; (b) After deferral.

or RM schedule. However, as a result, it can only reduce power consumption under fortunate circumstances when a sufficiently long idle interval occurs in the normal execution of the tasks.

3.4.2 Work-Idle-Conserving Schedulers

To improve the energy savings of the previous scheduler, one can try to increase the duration of idle periods to allow longer intervals in low-power mode and amortize switching-time costs over longer periods. However, care must be taken to ensure that no task will be delayed and miss its deadline.

A class of *work-idle-conserving* schedulers can help increase such idle durations. While there are tasks to execute, these schedulers follow the standard work-conserving RM or EDF scheduling policy. However, once all tasks have completed and the system enters idle, these schedulers become "idle-conserving" — they attempt to lengthen the idle period by deferring the next arriving task. This must be done conservatively to ensure future deadlines are not violated.

To this end, a simple algorithm for execution deferral looks ahead to the next arriving task, k, and will limit its effects to just this one task. Task k will arrive at time D_1, the earliest deadline in the system. Between time D_1 and time D_2, the next deadline in the system, task k will execute exclusively. If the WCET of task k, C_k, is less than D_2 −

D_1, then we can defer the starting time of task k by $D_2 - D_1 - C_k$ without affecting its deadline or the execution of any other task. This is illustrated in Figure 3.4. Since C_k time is available before D_2, task k still completes all execution by D_2 as with the unaltered schedule. Furthermore, the effects of this deferral are local to the interval (D_1, D_2), so no other task's execution is affected by the deferral of task k.

There are two caveats when implementing this algorithm. First, it is possible that two tasks have coinciding deadlines at time D_1. In this case, two tasks are released simultaneously, and we should not attempt to defer execution based on an algorithm that assumes just one task executes exclusively after D_1. This case is handled by simply using $D_2 = D_1$ in case of coinciding deadlines. Since $C_k > 0$ for either task, no deferral is performed. The second issue is that the second deadline, D_2, may actually be for the invocation of task k released at time D_1. At the time idle begins (before time D_1), this invocation of task k has not yet been released, and its deadline has not yet been added to the system, so this case must be checked when computing $D_2 - D_1$. Figure 3.5 presents the power-down algorithm for the simple work-idle-conserving RM/EDF scheduler. As before, in the case of RM, it may be necessary to add structures to keep track of task deadlines.

Essentially, this algorithm conservatively extends the previous algorithm to allow the deferral of execution for a single task in an attempt to extend idle intervals. Although this will improve performance over the simple RM/EDF scheduling with power-down described earlier, there is no guarantee that the deferral of the next task alone will provide greatly improved power-down time, particularly if tasks use significantly less than their WCETs.

3.4.3 Slack-Stealing Schedulers for Power-down

When tasks consume less than their WCETs, one would like to use the surplus time, or *slack*, as effectively as possible for power-down. However, with the simple approach of task deferral shown above, the slack is not directly taken into account, so a somewhat conservative mechanism is used to ensure that future deadlines are not violated. If one could accurately track the slack gained due to tasks completing early, then more aggressive deferral of task execution can be employed, while still ensuring that future deadlines are met.

In this next approach, called *slack-stealing scheduling for power-down*, the goal is to

Assume n tasks, sorted by deadline:

$D_1 \leq D_2 \leq \cdots \leq D_n$

/* this is already needed for EDF */

upon task_release(task i):

set $done_i$ to 0;

update D_i to $D_i + t_i$;

resort task list by new deadlines;

/* schedule by RM/EDF priority */

upon task_completion(task i):

set $done_i$ to 1;

if (for all j, $done_j$=1) then:

t_{now} = get_current_time();

t_{defer} = max{ 0, min{ $D_2 - D_1 - C_1, t_1 - C_1$ } }

if ($(D_1 + t_{defer} - t_{now}) > (t_{down} + t_{up})$) then:

set wakeup timer to $D_1 + t_{defer} - t_{now} - t_{up}$;

start power-down;

else idle;

else:

/* schedule by RM/EDF priority */

Figure 3.5: Work-idle-conserving scheduler

maintain an accurate count of the extra computing time (i.e., slack), and use this to generate longer idle intervals in the execution schedule. Existing slack-stealing techniques [42] use slack to provide time to real-time aperiodic tasks, increased execution time to variable runtime tasks, e.g., increasing rewards for increasing service (IRIS) [13], or to execute best-effort, non-real-time tasks. In this case, the computed slack time is used to determine the maximum period over which one can delay the execution of tasks (i.e., stay in an idle-coserving mode) to ensure the execution starting time is not delayed beyond that in the EDF or RM execution schedule assuming WCETs for all tasks. As long as the starting time for any part of a task occurs no later than in the EDF or RM WCET schedule, then task completion no later than in the WCET schedule is guaranteed, and, therefore, all deadlines are ensured to be met. There is only one exception to the policy of executing tasks no later than in the WCET schedule: as in the previous work-idle-conserving schedulers, single task deferral is also applied, when possible, which, as discussed earlier, will not cause any task to violate its deadlines.

As in the previous approach, the scheduling first proceeds in a work-conserving fashion. When an idle period is reached, the mode is switched to idle-conserving, and tasks that arrive in the future are deferred in a non-work-conserving manner. Once execution of a delayed task begins, the scheduler resumes work-conserving operation. While tasks are executing, the execution schedule, assuming WCETs for all tasks, is computed, so when idle occurs, the actual slack relative to the WCET schedule can be computed. This is used to determine the maximum duration over which the system may be powered down to ensure arriving tasks will begin no later than they would have in the WCET schedule. This is illustrated in the example in Figure 3.6.

The actual algorithm for slack-stealing EDF and RM scheduling is outlined in Figure 3.7. There are two general functions performed in the algorithm. First, a set of data structures is maintained that simulate the execution timing under the EDF or RM scheduler assuming WCETs for all tasks. These structures are updated while tasks are executed in a work-conserving manner. One should note that the release times of tasks in the actual execution and the simulated schedule are identical, but in the actual execution, tasks will complete earlier than in the WCET schedule. The second function is triggered when some task completes and no further work is immediately available. Then, the algorithm simu-

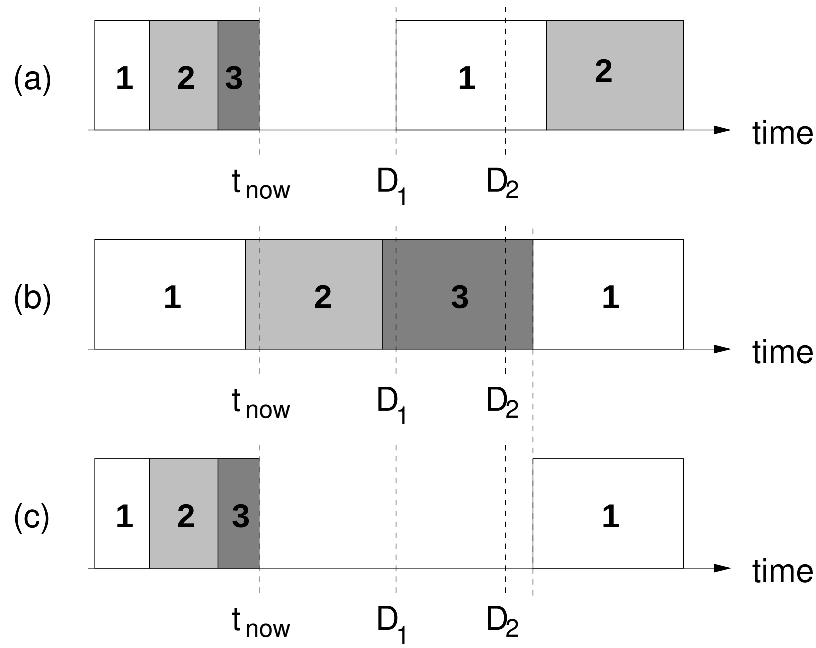


Figure 3.6: Slack-stealing scheduler example scenario. t_{now} is current time, where system enters idle. (a) Execution schedule for work-conserving scheduler indicates execution resumes at time D_1 when task 1 is released; (b) Canonical schedule assuming tasks always use exactly their WCETs indicates next invocation of task 1 would start after time D_2; (c) Slack-stealing power-down schedulers defer task 1 until the time indicated by the WCET schedule.

lates the continued execution of the WCET schedule forward in time, including the future releases of tasks, and determines the earliest time at which the WCET schedule indicates that a currently unreleased task commences execution. The difference between this time and the current time essentially constitutes the available slack due to tasks using less than their WCETs. The power-down interval is selected to terminate at this future time, or at the deferred start time computed by the previously-described work-idle-conserving mechanism, whichever is later. The system will then resume work-conserving execution until the next idle interval.

With this deferral technique, the system can guarantee timely execution of tasks (i.e., all tasks complete by their deadlines) by ensuring that execution occurs no later than in the WCET schedule, and that any greater deferral is limited to a single task with local effects that do not extend beyond any deadline (i.e., the work-idle-conserving mechanism

Assume n tasks, sorted by deadline:

    $D_1 \leq D_2 \leq \cdots \leq D_n$

```
upon task_completion(task i):
    simulate_execution();
    set done_i to 1;
    if (for all j, done_j=1) then:
        t_now = get_current_time();
        simulate_forward();
        t_resume = max{ t_sim, min{ D_2 - C_1, D_1 + t_1 - C_1 }}
        if ( (t_resume - t_now) > (t_down + t_up) ) then:
            set wakeup timer to t_resume - t_now - t_up;
            start power-down;
        else idle;
    else:    /* schedule by RM/EDF priority */

simulate_release(task i):
    set done_i to 0;
    set cc_i to C_i;
    update D_i to D_i + t_i;
    resort task list by new deadlines;
```

```
simulate_execution():
    t_now = get_current_time();
    repeat while t_sim < t_now:
        find task k such that cc_k ≠ 0 and for all j, D_j < D_k implies cc_j = 0
            /* for RM, replace D_j < D_k with t_j < t_k in line above */
        if k exists, then:
            t_run = min{ cc_k, t_now-t_sim, D_1-t_sim };
            set cc_k to cc_k − t_run;
        else t_run = min{ t_now-t_sim, D_1-t_sim };
        set t_sim to t_sim + t_run;
        for all j such that D_j ≤ t_sim, simulate_release( task j );

simulate_forward():
    loop:
        find task k such that cc_k ≠ 0 and for all j, D_j < D_k implies cc_j = 0
            /* for RM, replace D_j < D_k with t_j < t_k in line above */
        if k exists, then:
            if done_k = 0 jump out of loop;
            t_run = min{ cc_k, D_1-t_sim };
            set cc_k to cc_k − t_run;
        else t_run = D_1-t_sim;
        set t_sim to t_sim + t_run;
        for all j such that D_j ≤ t_sim, simulate_release( task j );
    end of loop;
```

Figure 3.7: Slack-stealing scheduler for power-down

described in previous section). Hence, the schedulability of the system and deadline guarantees are identical to the system with ordinary RM or EDF scheduling. This algorithm, although still an example of a bimodal work-idle-conserving scheduler, is more aggressive and has greater time scope than the previous schedulers, as it does permit deferral beyond multiple deadlines, allowing the deferral of multiple ready tasks.

3.4.4 Improved Slack-Stealing EDF

The slack-stealing EDF scheduler works well when the WCET schedule indicates a greatly deferred start time for tasks released in the future due to continued worst-case execution of currently released tasks. When the system is heavily-loaded (i.e., very little idle time in WCET schedule), this method can help greatly. However, when the system is lightly-loaded, there will be idle periods in the WCET schedule, which, as the simulated schedule is work-conserving, may greatly limit the deferral time and, consequently, the power-down intervals.

This next approach modifies the slack-stealing EDF scheduler slightly to improve the deferral time and power-down intervals when the system is under-utilized. The goal is to defer task execution more than the WCET EDF schedule would indicate, but still ensure deadlines of the tasks. This is accomplished by creating a specification of an alternate task set that fully utilizes the system, and using this for the simulation of the WCET schedule. For each actual task i, there is a task i' in the alternate set with an identical period, t_i. The WCET of task i', C_i' is such that $C_i' \geq C_i$. Since the period, and therefore deadlines, of task i' are identical to those of the real task i, and since the WCET is at least as long as for i, any schedule that can guarantee the timely execution of task i' will also suffice for i. As this is true for all tasks, as long as the alternate task set is schedulable, so is the real task set using the same execution schedule.

To create a schedulable alternate task set for an EDF scheduler, assuming negligible preemption and scheduler overheads, one needs to simply ensure that the total utilization does not exceed 1, i.e., $\sum_i C_i/t_i \leq 1$ [45]. Figure 3.8 shows the algorithm to generate a schedulable alternate task set that fully utilizes the system. First, the worst-case utilization of the given task set is computed as U. For an under-utilized system, this value is strictly bounded, $0 < U < 1$. For a fully-utilized system, $U = 1$, so the alternate task set is con-

Assume n tasks

Each task i has period t_i and WCET C_i

At startup:

  Compute $U = \sum_i C_i/P_i$

  For each task i:

    create alternate task i'

    set t'_i to t_i

    set C'_i to C_i/U

Proceed with slack-stealing EDF scheduling

  but use alternate task set for simulated WCET schedule

Figure 3.8: Improved slack-stealing EDF scheduler for power-down

structed to ensure this. Each alternate task is given the same period as its real counterpart, but its WCET is multiplied by a factor of $1/U$. Now, this alternate task set is used for the simulated WCET schedule in the slack-stealing EDF scheduler. As this alternate schedule has the same deadlines and greater execution time available for each task than needed for the given task set, ensuring all tasks execute no later than in this alternate WCET schedule suffices to guarantee task deadlines. Again, the one exception to starting a task no later than in the WCET schedule is when the work-idle-conserving EDF scheduler's single task deferral is applied, but its effects are localized to the single task and do not cross deadlines, so task deadline guarantees are maintained.

By using an alternate task set that fully utilizes the system, has the same deadlines, and has greater execution time required for each task than the given task set as the reference for task start times, greater deferral times, longer power-down intervals, and lower energy consumption can be achieved by the improved slack-stealing EDF algorithm.

3.4.5 Handling Multiple Power-down States

The sprint-and-halt algorithms as discussed support hardware with two power states: active and power-down. However, often, there may be multiple power-down states available on a platform. These states will have varying power-consumption rates, as well as latencies to resume active-state operation. For example, a system may have a low-latency processor halt mechanism that moderately reduces power consumption, as well as a system power-down mode that greatly reduces power, at the expense of a longer resume latency.

It is possible to modify all of the algorithms introduced here to support more than one power-down state. First, one needs to determine the power model of the system with two or more power-down states. To keep everything consistent, assume simply that each power state x has its own constant power-consumption rate, $P_{low,x}$. Furthermore, only transitions between each low-power state and the active state are considered, i.e., do not transition from one power-down mode to another directly. Each state x has transition latencies $t_{up,x}$ and $t_{down,x}$ to switch to and from active state. During the transitions, an average power of $P_{trans,x}$ is dissipated.

Given this model, the algorithms need to choose the power state that requires the lowest energy cost for any power-down interval t_{pd}. The energy consumed by state x, E_x, can be expressed as a function of the power-down interval, t_{pd}:

$$E_x(t_{pd}) = (t_{down,x} + t_{up,x})P_{trans,x} + (t_{pd} - t_{down,x} - t_{up,x})P_{low,x}$$

Assuming two states, x and y, where $(t_{down,x}+t_{up,x}) < (t_{down,y}+t_{up,y})$ and $P_{low,x} > P_{low,y}$, i.e., y is a lower-power, longer-latency state than x, it is better to switch to state y when $E_x(t_{pd}) > E_y(t_{pd})$. Solving this for t_{pd}, it is better to use state y when:

$$t_{pd} > \frac{(t_{down,y} + t_{up,y})(P_{trans,y} - P_{low,y}) - (t_{down,x} + t_{up,x})(P_{trans,x} - P_{low,x})}{P_{low,x} - P_{low,y}}.$$

The right-hand side of the inequality depends only on system parameters, so it can be computed ahead of time and used as a constant, called $t_{E_x=E_y}$. So, to support two power-down states, the algorithms simply decide based on the power-down interval:

$$\begin{aligned} \text{idle} &: \quad t_{pd} < (t_{down,x} + t_{up,x}) \\ \text{state } y &: \quad t_{pd} > \max\{(t_{down,y} + t_{up,y}), t_{E_x=E_y}\} \\ \text{state } x &: \quad \text{otherwise} \end{aligned}$$

With a larger number of low-power states, one can similarly use the energy computation above and solve the inequality for t_{pd} to determine the range of t_{pd} for which one state is better than another. Using all such boundary values of t_{pd} for all possible pairs of states, and the minimum t_{pd} that allows the use of each state, one can find a simple static mapping from t_{pd} to the power state that results in the lowest energy dissipation.

3.5 Evaluation

To evaluate the potential energy savings provided by the various sprint-and-halt scheduling algorithms described above, one can use a system simulation to predict energy dissipation across a broad range of scenarios. The following subsection describes the simulator developed to evaluate the power-down scheduling techniques and the assumptions made in its design. Following this, some simulation results are presented to provide insight into the system parameters affecting energy savings in a real-time system with software-controlled power-down capabilities.

3.5.1 Simulation Methodology

The sprint-and-halt algorithms are evaluated using a simulator developed using C++ that models the operation of hardware capable of software-controlled power-down under a wide range of system characteristics. The simulator takes as input a task set, specified with the period and computation requirements of each task, as well as several system parameters, and provides the energy consumption of the system for each of the algorithms presented earlier. Real-time schedulers without any power-down support are also simulated for comparison. Parameters supplied to the simulator include the hardware specification, i.e., P_{high}, P_{low}, P_{trans}, t_{down}, and t_{up}, and a specification of the fraction of the WCET that the tasks should actually consume. This latter parameter can be a constant (e.g., 0.9 indicates that each task will use 90% of its specified worst-case computation cycles during each invocation), or can be a random function (e.g., uniformly-distributed random multiplier for each invocation).

The simulation assumes the bimodal system power model described in Section 3.3. Each simulated cycle used for task execution or idle consumes a constant energy quantum, derived from P_{high}, while each cycle in the power-down state consumes energy based on P_{low}. Although the simulator can use arbitrary P_{trans} values, the evaluations presented here assume the worst-case situation where $P_{trans} = P_{high}$. With this model, variations due to different types of instructions executed are not taken into account. This simplification eliminates the need for actual execution traces, and a simpler cycle counting approach can be used to determine energy consumption. The simulator only considers the time/energy overheads of switching into and out of the power-down state. In particular, it does not consider preemption and task-switch overheads, or the overheads of executing scheduler code. However, these are small relative to the range of power-state switching latencies considered, so there is no loss of generality from these assumptions. Besides, the relative energy performance of different scheduling algorithms will not be affected by this assumption.

The real-time task sets are specified using a pair of numbers for each task, indicating its period and worst-case execution time. The task sets are generated randomly as follows. Each task has an equal probability of having a short (1–10 ms), medium (10–100 ms), or long (100–1000 ms) period. Within each range, task periods are uniformly distributed. This simulates the varied mix of short- and long- period tasks commonly found in real-time systems. The computation requirements of the tasks are assigned randomly using a similar 3-range uniform distribution. Finally, the task computation requirements are scaled by a constant chosen such that the sum of the utilizations of the tasks in the task set reaches a desired value. This method of generating real-time task sets has been used previously in the development and evaluation of a real-time embedded microkernel [95]. Averaged across hundreds of distinct task sets generated for several different total worst-case utilization values, the simulations provide a relationship of energy consumption to the worst-case utilization of the task sets, or to the power-down-power-up latencies.

3.5.2 Results

The simulator described above permits the energy consumption comparison of the sprint-and-halt schedulers to each other as well as against real-time scheduling without power-down support. In addition, the schedulers are also compared to a theoretical lower bound on energy. This lower bound is computed based on the observation that the highest frequency task in the system limits the maximum duration of the power-down state.

Considering just this one task, and assuming actual execution times are known, then it is possible to execute this task as late as possible so it completes exactly at its deadline, and then its next invocation is released and executed immediately. The system can power down until sufficiently before the following invocation's deadline to execute the task. As a result, at best, one power-down interval can span at most 2 periods of the highest frequency task. The lower bound energy is computed assuming all of the idle time is lumped together and divided into power-down intervals exactly 2 times the length of the period of the highest frequency task. All of the execution time is likewise lumped together at the start of the lower-bound simulation. This lower bound is in practice unachievable, but does give some insight into limitations on further energy improvements. We limit the experiments to the EDF versions of the sprint-and-halt algorithms.

Effects of power-down specifications: The first set of experiments determines the effects of varying the specifications of the power-down hardware. Figure 3.9 shows the energy dissipation for each sprint-and-halt EDF-based algorithm, normalized with respect to EDF scheduling without power-down support. Here, the task sets all contain 8 random tasks as described earlier, such that the total worst-case processor utilization is 0.95. The tasks' actual execution times are fixed to WCET/3. The average energy for the task sets is plotted for varying values of power-down latency, i.e., $t_{down} + t_{up}$, which are shown on a log scale. The three separate plots correspond to different P_{high}/P_{low} ratios.

One should immediately note that all of the algorithms can potentially save significant amounts of energy, particularly when the power-state transition latencies are small. In addition, the actual ratio of P_{high} to P_{low} does not affect the relative performance of the schedulers or the general trend of the curves significantly. Only the maximal achievable savings is affected. Finally, at the extreme range of power-state transistion latencies, the algorithms perfrom very close to the computed lower bound on energy. Between these extremes, the improved slack-stealing EDF (SS EDF+) scheduler performs the best, followed by slack-stealing EDF (SS EDF) and work-idle-conserving EDF (WIC EDF) schedulers. Even the simple EDF with power-down (EDF+PD) scheduler performs much better than plain vanilla EDF scheduling.

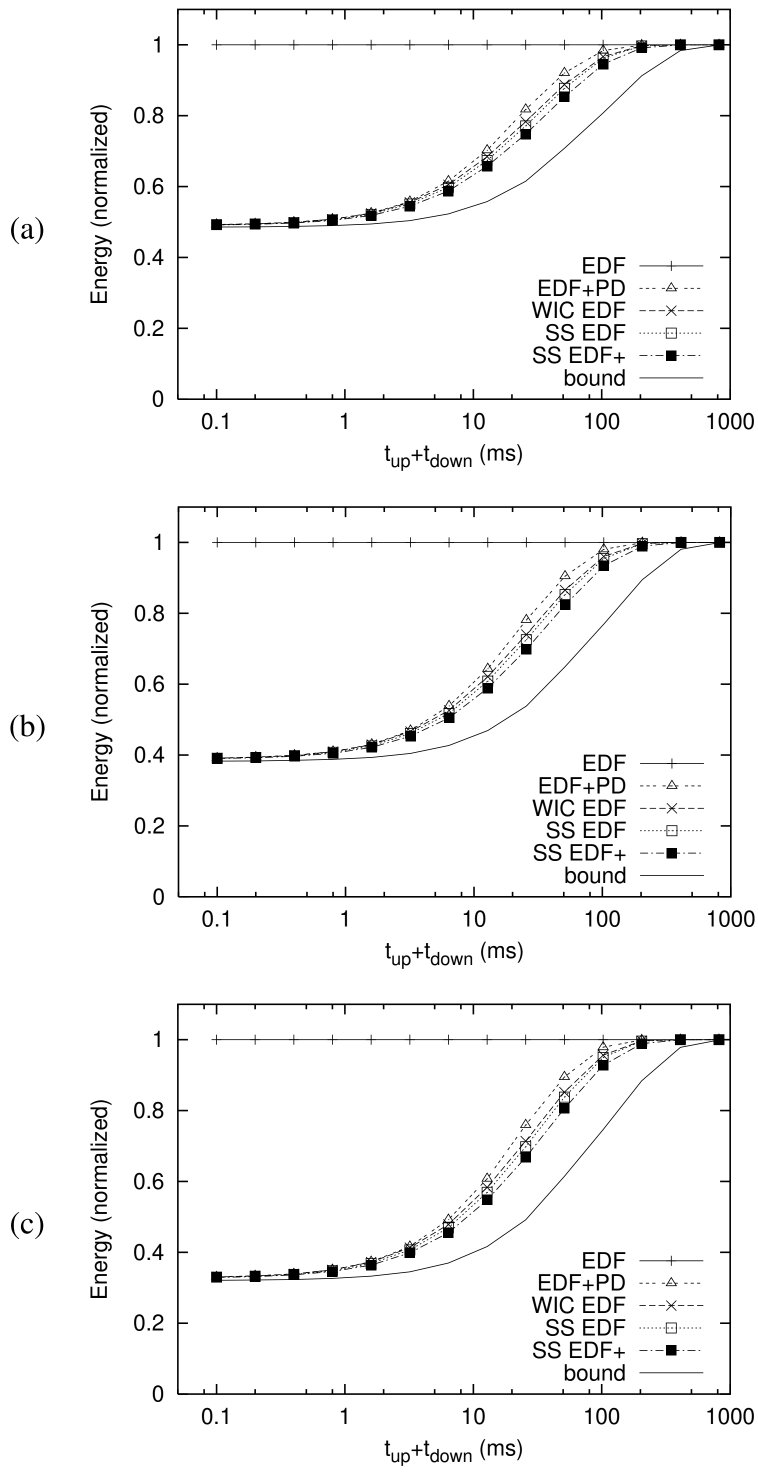
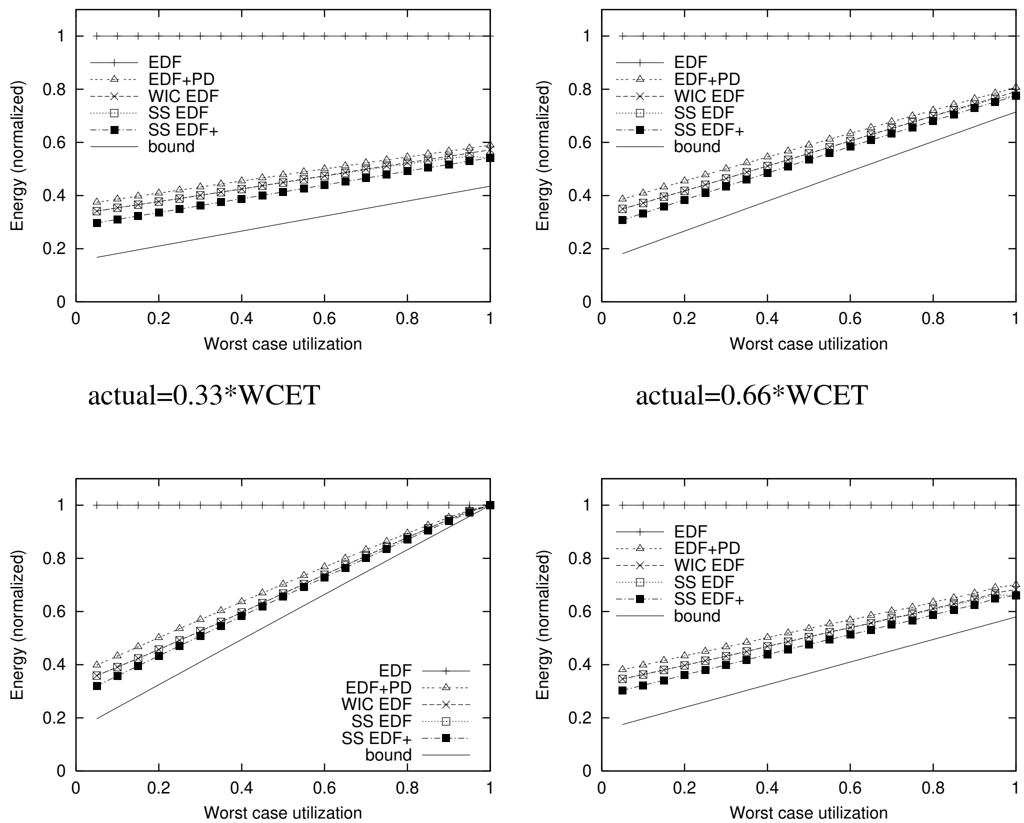


Figure 3.9: Effects of varying power-down hardware specifications: (a) $P_{high}/P_{low} = 4$; (b) $P_{high}/P_{low} = 10$; (c) $P_{high}/P_{low} = 100$



actual=0.33*WCET actual=0.66*WCET

actual=1.00*WCET actual=uniform(0,WCET)

Figure 3.10: Effects of varying workload parameters

Effects of workload parameters: The next set of experiments fix the harware specification and vary instead the task set parameters. In Figure 3.10, P_{high}/P_{low} is set to 20, and $t_{down} + t_{up}$ set to 10 ms. There are still 8 tasks in each random task set, but now the worst-case utilization of the task sets is varied, and resulting average energy across the task sets plotted. The actual execution times for the task is also varied, set to 0.33, 0.66, and 1.0 times the WCET for the first three subplots. The fourth plot uses a unform random distribution for the actual execution times of each task invocation.

Overall, the average energy profiles of the schedulers across multiple random task sets seem to vary fairly linearly with the worst-case utilization of the task sets. The one interesting exception is the simple slack-stealing EDF (SS EDF) scheduler. For most of the range of utilization, it performs nearly identically to the work-idle-conserving (WIC EDF) scheduler. However, at very high worst-case utilizations, it performs better than WIC EDF. This is due to the fact that at high utilizations, WCET EDF schedules have very little idle time, so the algorithm, attempting to pace execution to the WCET schedule, is capable of much longer deferrals. It is this very effect that motivates the improved slack-stealing approach (SS EDF+).

The change in actual execution times affects the slope of the average energy response as worst-case utilization is varied. Using uniformly-distributed random execution times (between 0 and WCET for each task invocation) does not significantly change the average energy curves. As the average execution times are generally much smaller than the WCETs for most real-time task sets, the first plot is closest to what one can deem as typical. Here, with the reasonable assumption of 20:1 P_{high} to P_{low} ratio and 10 ms power-down latency, the sprint-and-halt schedulers achieve 40–70% energy reduction over EDF without power-down support.

Relative performance of power-down schedulers: Although all of the power-down schedulers perform much better than ordinary EDF without power-down support, it is interesting to see how well the more complex techniques perform relative to the simple EDF with power-down added. Figure 3.11 shows this relationship, assuming actual task execution times are 0.33*WCET. The first plot indicates that with a power-down latency of 10 ms, the improved slack-stealing approach reduces average energy by approximately 10–20% over EDF+PD as the task set worst case processor utilizations vary. In the second plot, the worst-case utilization is fixed to 0.95, and the power-down latency is varied across the log scale. At the extremes, there is very little improvement over EDP+PD. However, in the middle of the range, where the power-down latency is comparable to the task periods, the more advanced techniques show up to 10% energy reduction relative to the simple EDF+PD scheduler.

3.6 Related Work

Reducing power consumption in mobile devices is a very active area of research. The authors of [5, 49] enumerate and survey a wide variety of approaches to energy reduction on mobile platforms from a high-level perspective. A variety of techniques for powering down subsystems, including display backlight, disk drives, and communication channels,

(a)

(b)

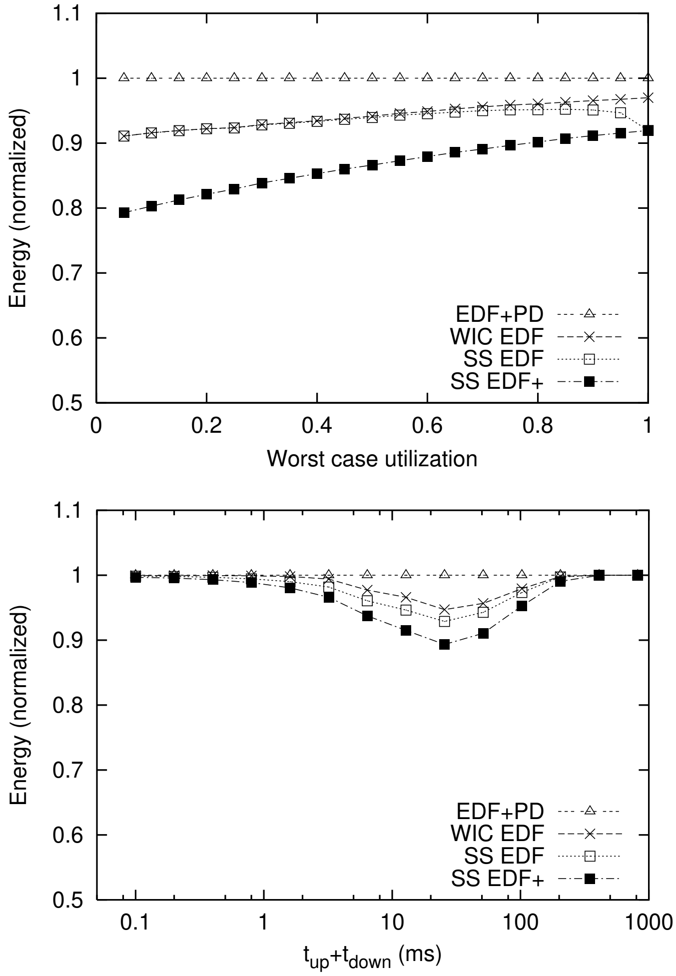


Figure 3.11: Relative performance of sprint-and-halt algorithms: (a) $t_{up} + t_{down}$=10 ms; (b) Worst case utilization=0.95

and applying circuit tricks to reduce power are cited. However, little work has been done in the context of powering down systems (including the processor) when real-time constraints are present.

Although not intended for real-time systems, the authors of [27] developed techniques for powering down systems that execute event-driven (i.e., user-interactive) applications. Using idle history, future idle durations are predicted. This, unfortunately, cannot provide the timing guarantees needed for real-time systems. This work also proposes a *pre-wake* technique that reactivates the system early to compensate for wake-up latencies and improve responsiveness. A similar mechanism is used in all of the sprint-and-halt algorithms, as this is necessary in order to ensure deadline guarantees.

Focusing on powering down I/O subsystems, the authors of [50] attempted to maximize the effectiveness of power-down by increasing the duration and amortizing switching overheads. This is similar in concept to the sprint-and-halt algorithms, but is meant for I/O, not the processor. Furthermore, it involves reordering tasks to coalesce common device accesses, which would affect timings of tasks and preclude its use in real-time systems.

Extending this to real-time systems, the authors of [83] presented a device power-scheduling algorithm that preserves real-time guarantees. Deadlines are preserved by keeping the task execution schedule unaltered and fitting power-down events for I/O devices whenever possible. In contrast, sprint-and-halt scheduling does alter the task execution schedule, while preserving deadlines, and can power down the processor.

Finally, there has been much research on dynamic voltage scaling of the CPU to conserve energy since the earliest papers on this topic appeared [6, 23, 90]. These mechanisms have also been extended to work in real-time systems [24, 59, 65]. DVS algorithms try to execute tasks as slowly as possible to spread out work and eliminate idle time, while in contrast, sprint-and-halt techniques try to coalesce work and execute it as fast as possible to allow longer power-down intervals. Hence, these real-time scheduling algorithms approach energy conservation with directly opposite philosophies.

3.7 Conclusions

This chapter has presented a class of sprint-and-halt scheduling algorithms that attempt to make best use of software-controlled power-down to reduce energy expenditure, while meeting hard, real-time constraints. Several algorithms of increasing complexity have been developed to better amortize energy costs due to the transition latencies to and from low-power states. Extensive simulations show that with some typical system parameters, the power-down techniques can save 40–70% of the energy dissipated in an unmodified system, while preserving all real-time deadline guarantees. Sensitivity experiments show that for very large (100's of ms) and very small (100's of μs) power-down latencies, the simplest power-down scheduling techniques suffice, as all of the methods approach a theoretical lower bound. However, for moderate power-down latencies, the advanced techniques provide 10–20% lower energy consumption relative to the simplest sprint-and-halt schedulers.

Future research directions related to this work include extending sprint-and-halt scheduling to less restrictive real-time paradigms. A probabilistic real-time approach may provide greater flexibility in using power-down techniques and allow greater energy savings. Integration of sprint-and-halt with other power-reduction techniques, such as dynamic voltage scaling, may also be possible in a hybrid solution that switches between the two depending on workload characteristics and available idle time.

CHAPTER 4

Exploiting Hardware Energy-Conservation Mechanisms through Real-Time Dynamic Voltage Scaling (RT-DVS)

This chapter, as did the previous chapter, considers how system software can maximally exploit energy-conserving features of the underlying hardware platform, while adhering to timeliness and real-time application constraints. In particular, Dynamic Voltage Scaling (DVS) has been a key technique in exploiting the hardware characteristics of processors to reduce energy dissipation by lowering the supply voltage and operating frequency. DVS algorithms have demonstrated dramatic energy savings while providing the necessary peak computation power in general-purpose systems, but for a large class of applications in embedded real-time systems like cellular phones and camcorders, the variable operating frequency interferes with real-time deadline guarantee mechanisms, and DVS in this context, despite its growing importance, is largely overlooked/under-developed. To provide real-time guarantees, DVS must consider deadlines and periodicity of real-time tasks, requiring integration with the real-time scheduler. This chapter presents a class of novel algorithms, called *real-time DVS* (RT-DVS), that modifies the OS's real-time scheduler and task management service to provide significant energy savings while maintaining real-time deadline guarantees. Through simulations and a working prototype implementation, these RT-DVS algorithms are shown to closely approach the theoretical lower bound on energy consumption, and can easily reduce energy consumption 20% to 40% in an embedded real-time system.

4.1 Introduction

Computation and communication have been steadily moving toward mobile and portable platforms/devices. This is very evident in the growth of laptop computers and PDAs, but is also occurring in the embedded world. With continued miniaturization and increasing computation power, we see ever growing use of powerful microprocessors running sophisticated, intelligent control software in a vast array of devices including digital camcorders, cellular phones, and portable medical devices.

Unfortunately, there is an inherent conflict in the design goals behind these devices: as mobile systems, they should be designed to maximize battery life, but as intelligent devices, they need powerful processors, which consume more energy than those in simpler devices, thus reducing battery life. In spite of continuous advances in semiconductor and battery technologies that allow microprocessors to provide much greater computation per unit of energy and longer total battery life, the fundamental tradeoff between performance and battery life remains critically important.

Recently, significant research and development efforts have been made on *Dynamic Voltage Scaling* (DVS) [6, 20, 23, 24, 36, 47, 56, 59, 60, 62, 68, 69, 82, 90]. DVS tries to address the tradeoff between performance and battery life by taking into account two important characteristics of most current computer systems: (1) the peak computing rate needed is much higher than the average throughput that must be sustained; and (2) the processors are based on CMOS logic. The first characteristic effectively means that high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance, low-power processor would suffice. We can achieve the low performance by simply lowering the operating frequency of the processor when full speed is not needed. DVS goes beyond this and scales the operating voltage of the processor along with the frequency. This is possible because static CMOS logic, used in the vast majority of microprocessors today, has a voltage-dependent maximum operating frequency, so when used at a reduced frequency, the processor can operate at a lower supply voltage. Since the energy dissipated per cycle with CMOS circuitry scales quadratically to the supply voltage ($E \propto V^2$) [6], DVS can potentially provide a very large net energy savings through frequency and voltage scaling.

In time-constrained applications, often found in embedded systems like cellular phones and digital video cameras, DVS presents a serious problem. In these real-time embedded systems, one cannot directly apply most DVS algorithms known to date, since changing the operating frequency of the processor will affect the execution time of the tasks and may cause the violation of timeliness guarantees. In this chapter, we present several novel algorithms that incorporate DVS into the OS scheduler and task management services of a real-time embedded system, providing the energy savings of DVS while preserving deadline guarantees. This is in sharp contrast with the average throughput-based mechanisms typical of many current DVS algorithms. In addition to detailed simulations that show the energy-conserving benefits of our algorithms, we also present an actual implementation of our mechanisms, demonstrating them with measurements on a working system. To the best of our knowledge, this is one of the first working implementations of DVS, and the first implementation of *Real-Time DVS* (RT-DVS).

In the next section, we present details of DVS, real-time scheduling, and our new RT-DVS algorithms. Section 3 presents the simulation results and provides insight into the system parameters that most influence the energy-savings potential of RT-DVS. Section 4 describes our implementation of RT-DVS mechanisms in a working system and some measurements obtained. Section 5 presents related work and puts our work in a larger perspective before we close with our conclusions and future directions in Section 6.

4.2 RT-DVS

To provide energy-saving DVS capability in a system requiring real-time deadline guarantees, we have developed a class of RT-DVS algorithms. In this section, we first consider DVS in general, and then discuss the restrictions imposed in embedded real-time systems. We then present RT-DVS algorithms that we have developed for this time-constrained environment.

4.2.1 Why DVS?

Power requirements are one of the most critical constraints in mobile computing applications, limiting devices through restricted power dissipation, shortened battery life, or

Screen	CPU subsystem	Disk	Power
On	Idle	Spinning	13.5 W
On	Idle	Standby	13.0 W
Off	Idle	Standby	7.1 W
Off	Max. Load	Standby	27.3 W

Table 4.1: Power consumption measured on Hewlett-Packard N3350 laptop computer

increased size and weight. The design of portable or mobile computing devices involves a tradeoff between these characteristics. For example, given a fixed size or weight for a handheld computation device/platform, one could design a system using a low-speed, low-power processor that provides long battery life, but poor performance, or a system with a (literally) more powerful processor that can handle all computational loads, but requires frequent battery recharging. This simply reflects the cost of increasing performance — for a given technology, the faster the processor, the higher the energy costs (both overall *and* per unit of computation).

The discussion in this chapter will generally focus on the energy consumption of the processor in a portable computation device for two main reasons. First, the practical size and weight of the device are generally fixed, so for a given battery technology, the available energy is also fixed. This means that only power consumption affects the battery life of the device. Secondly, we focus particularly on the processor because in most applications, the processor is the most energy-consuming component of the system. This is definitely true on small handheld devices like PDAs [18], which have very few components, but also on large laptop computers [48] that have many components including large displays with backlighting. Table 4.1 shows measured power consumption of a typical laptop computer. When it is idle, the display backlighting accounts for a large fraction of dissipated power, but at maximum computational load, the processor subsystem dominates, accounting for nearly 60% of the energy consumed. As a result, the design problem generally boils down to a tradeoff between the computational power of the processor and the system's battery life.

One can avoid this problem by taking advantage of a feature very common in most computing applications: the average computational throughput is often much lower than the peak computational capacity needed for adequate performance. Ideally, the processor would be "sized" to meet the average computational demands, and would have low energy costs per unit of computation, thus providing good battery life. During the (relatively rare) times when peak computational load is imposed, the higher computational throughput of a more sophisticated processor would somehow be "configured" to meet the high performance requirement, but at a higher energy cost per unit of computation. Since the high-cost cycles are applied for only some, rather than all, of the computation, the energy consumption will be lower than if the more powerful processor were used all of the time, but the performance requirements are still met.

One promising mechanism that provides the best of both low-power and high-performance processors in the same system is DVS [90]. DVS relies on special hardware, in particular, a programmable DC-DC switching voltage regulator, a programmable clock generator, and a high-performance processor with wide operating ranges, to provide this best-of-both-worlds capability. In order to meet peak computational loads, the processor is operated at its normal voltage and frequency (which is also its maximum frequency). When the load is lower, the operating frequency is reduced to meet the computational requirements. In CMOS technology, used in virtually all microprocessors today, the maximum operating frequency increases (within certain limits) with increased operating voltage, so when the processor is run slower, a reduced operating voltage suffices [6]. A second important characteristic is that the energy consumed by the processor per clock cycle scales quadratically with the operating voltage ($E \propto V^2$) [6], so even a small change in voltage can have a significant impact on energy consumption. By dynamically scaling both voltage and frequency of the processor based on computation load, DVS can provide the performance to meet peak computational demands, while on average, providing the reduced power consumption (including energy per unit computation) benefits typically available on low-performance processors.

4.2.2 Real-time issues

For time-critical applications, however, the scaling of processor frequency is often detrimental. Particularly in real-time embedded systems like portable medical devices and cellular phones, where tasks must be completed by some specified deadlines, most algorithms for DVS known to date cannot be applied. These DVS algorithms do not consider real-time constraints and are based on solely average computational throughput [23, 60, 90]. Typically, they use a simple feedback mechanism, such as detecting the amount of idle time on the processor over a period of time, and then adjust the frequency and voltage to just handle the computational load. This is very simple and follows the load characteristics closely, but cannot provide any timeliness guarantees and tasks may miss their execution deadlines. As an example, in an embedded camcorder controller, suppose there is a program that must react to a change in a sensor reading within a 5 ms deadline, and that it requires up to 3 ms of computation time with the processor running at the maximum operating frequency. With a DVS algorithm that reacts only to average throughput, if the total load on the system is low, the processor would be set to operate at a low frequency, say half of the maximum, and the task, now requiring up to 6 ms of processor time, cannot meet its 5 ms deadline. In general, none of the average throughput-based DVS algorithms found in literature can provide real-time deadline guarantees.

In order to realize the reduced energy-consumption benefits of DVS in a real-time embedded system, we need new DVS algorithms that are tightly-coupled with the actual real-time scheduler of the operating system. In the classic model of a real-time system, there is a set of tasks that need to be executed periodically. Each task, T_i, has an associated period, P_i, and a worst-case computation time, C_i.[1] The task is *released* (put in a runnable state) periodically once every P_i time units, at which point it can begin execution. The task needs to complete its execution by its deadline, typically defined as the end of the period [45], i.e., by the next release of the task. As long as each task T_i uses no more than C_i cycles in each invocation, a real-time scheduler can guarantee that the tasks will always receive enough processor cycles to complete each invocation on time. Of course, to provide such guarantees, there are some conditions placed on allowed task sets, often expressed in the form of *schedulability tests*. A real-time scheduler guarantees that tasks will meet their deadlines given that:

C1. the task set is *schedulable* (passes schedulability test), and

C2. no task exceeds its specified worst-case computation bound.

DVS, when applied in a real-time system, must ensure that both of these conditions hold.

[1] Although not explicit in the model, aperiodic and sporadic tasks can be handled by a periodic or deferred server [43] For non-real-time tasks, too, we can provision processor time using a similar periodic server approach.

In this chapter, we develop algorithms to integrate DVS mechanisms into the two most-studied real-time schedulers, *Rate Monotonic* (RM) and *Earliest-Deadline-First* (EDF) schedulers [37, 41, 44, 45, 80]. RM is a static priority scheduler, and assigns task priority according to period — from the tasks that are ready to run (released for execution), it always selects the task with the shortest period to run first. EDF is a dynamic priority scheduler that sorts tasks by deadlines and always gives the highest priority to the released task with the most imminent deadline. In the classical treatments of these schedulers [45], both assume that the task deadline equals the period (i.e., the task must complete before its next invocation), that scheduling and preemption overheads are negligible,[2] and that the tasks are independent (no task will block waiting for another task). In our design of DVS for real-time systems, we maintain the same assumptions, since our primary goal is to reduce energy consumption, rather than to derive general scheduling mechanisms.

In the rest of this section, we present our algorithms that perform DVS in time-constrained systems without compromising deadline guarantees of real-time schedulers.

4.2.3 Static voltage scaling

We first propose a very simple mechanism for providing voltage scaling while maintaining real-time schedulability. In this mechanism we select the lowest possible operating frequency that will allow the RM or EDF scheduler to meet all the deadlines for a given task set. This frequency is set statically, and will not be changed unless the task set is changed.

To select the appropriate frequency, we first observe that scaling the operating frequency by a factor α ($0 < \alpha \leq 1$) effectively results in the worst-case computation time needed by a task to be scaled by a factor $1/\alpha$, while the desired period (and deadline) remains unaffected. We can take the well-known schedulability tests for EDF and RM schedulers from the real-time systems literature, and by using the scaled values for worst-case computation needs of the tasks, can test for schedulability at a particular frequency. The necessary and sufficient schedulability test for a task set under ideal EDF scheduling requires that the sum of the worst-case *utilizations* (computation time divided by period)

[2] We note that one could account for preemption overheads by computing the worst-case preemption sequences for each task and adding this overhead to its worst-case computation time.

EDF_test (α):

if ($C_1/P_1 + \cdots + C_n/P_n \leq \alpha$) return true;

else return false;

RM_test (α):

if ($\forall T_i \in \{T_1, \ldots, T_n | P_1 \leq \cdots \leq P_n\}$

$\lceil P_i/P_1 \rceil * C_1 + \cdots + \lceil P_i/P_i \rceil * C_i \leq \alpha * P_i$)

return true;

else return false;

select_frequency:

use lowest frequency $f_i \in \{f_1, \ldots, f_m | f_1 < \cdots < f_m\}$

such that RM_test(f_i/f_m) or EDF_test(f_i/f_m) is true.

Figure 4.1: Static voltage scaling algorithm for EDF and RM schedulers

be less than one, i.e., $C_1/P_1 + \cdots + C_n/P_n \leq 1$ [45]. Using the scaled computation time values, we obtain the EDF schedulability test with frequency scaling factor α:

$$C_1/P_1 + \cdots + C_n/P_n \leq \alpha$$

Similarly, we start with the sufficient (but not necessary) condition for schedulability under RM scheduling [37] and obtain the test for a scaled frequency (see Figure 4.1). The operating frequency selected is the lowest one for which the modified schedulability test succeeds. The voltage, of course, is changed to match the operating frequency. Assume that the operating frequencies and the corresponding voltage settings available on the particular hardware platform are specified in a table provided to the software. Figure 4.1 summarizes the static voltage scaling for EDF and RM scheduling, where there are m operating frequencies $f_1, \ldots, f_m$ such that $f_1 < f_2 < \cdots < f_m$.

Figure 4.2 illustrates these mechanisms, showing sample worst-case execution traces under statically-scaled EDF and RM scheduling. The example uses the task set in Table 4.2, which indicates each task's period and worst-case computation time, and assumes that three

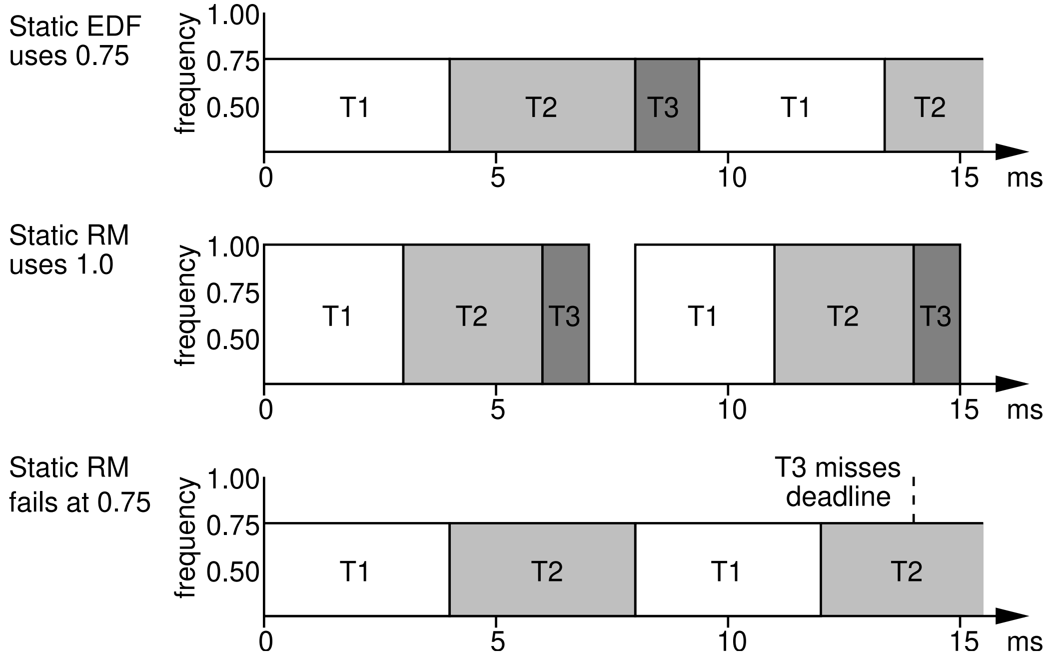


Figure 4.2: Static voltage scaling example

Task	Computing Time	Period
1	3 ms	8 ms
2	3 ms	10 ms
3	1 ms	14 ms

Table 4.2: Example task set, where computing times are specified at the maximum processor frequency

normalized, discrete frequencies are available (0.5, 0.75, and 1.0). The figure also illustrates the difference between EDF and RM (i.e., deadline vs. rate for priority), and shows that statically-scaled RM cannot reduce frequency (and therefore reduce voltage and conserve energy) as aggressively as the EDF version.

As long as for some available frequency, the task set passes the schedulability test, and as long as the tasks use no more than their scaled computation time, this simple mechanism will ensure that frequency and voltage scaling will not compromise the timely execution of tasks by their deadlines. The frequency and voltage setting selected are static with respect to a particular task set, and are changed only when the task set itself changes. As a result, this mechanism need not be tightly-coupled with the task management functions of the real-time operating system, simplifying implementation. On the other hand, this algorithm may not realize the full potential of energy savings through frequency and voltage scaling. In particular, the static voltage scaling algorithm does not deal with the most common situations where a task uses less than its worst-case requirement of processor cycles. To deal with this common situation, we need more sophisticated RT-DVS mechanisms.

4.2.4 Cycle-conserving RT-DVS

Although real-time tasks are specified with worst-case computation requirements, they generally use much less than the worst case on most invocations. To take best advantage of this, a DVS mechanism could reduce the operating frequency and voltage when tasks use less than their worst-case time allotment, and increase frequency to meet the worst-case needs. When a task is released for its next invocation, we cannot know how much computation it will actually require, so we must make the conservative assumption that it will need its specified worst-case processor time. When the task completes, we can compare the actual processor cycles used to the worst-case specification. Any unused cycles that were allotted to the task would normally (or eventually) be wasted, idling the processor. Instead of idling for extra processor cycles, we can devise DVS algorithms that avoid wasting cycles (hence "cycle-conserving") by reducing the operating frequency. This is somewhat similar to slack time stealing [42], except surplus time is used to run other remaining tasks at a lower CPU frequency rather than accomplish more work. These algorithms are tightly-coupled with the operating system's task management services, since they may need to reduce frequency on each task completion, and increase frequency on each task release. The main challenge in designing such algorithms is to ensure that deadline guarantees are not violated when the operating frequencies are reduced.

For EDF scheduling, as mentioned earlier, we have a very simple schedulability test: as long as the sum of the worst-case task utilizations is less than α, the task set is schedulable when operating at the maximum frequency scaled by factor α. If a task completes earlier than its worst-case computation time, we can reclaim the excess time by recomputing utilization using the actual computing time consumed by the task. This reduced value is used until the task is released again for its next invocation. We illustrate this in Figure 4.3, using the same task set and available frequencies as before. However, here, each invocation of the tasks may use less than the specified worst-case times, and follows the actual execution

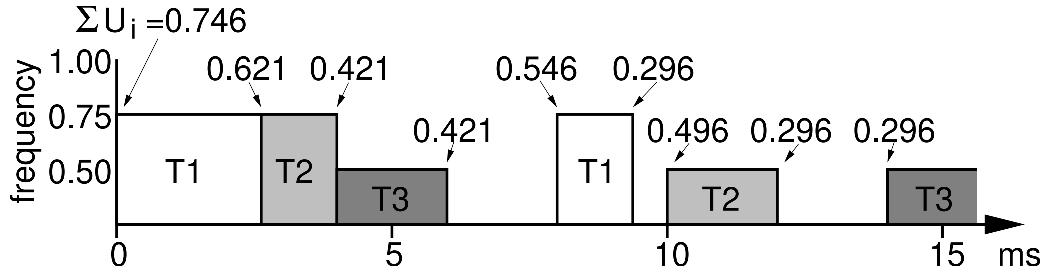


Figure 4.3: Example of cycle-conserving EDF

Task	Invocation 1	Invocation 2
1	2 ms	1 ms
2	1 ms	1 ms
3	1 ms	1 ms

Table 4.3: Actual computation requirements of the example task set (assuming execution at max. frequency)

times given in Table 4.3. As the actual execution requirements cannot be known to the system until after the task completes execution, at each scheduling point (task release or completion), the utilization is recomputed using the actual time for completed tasks and the specified worst case for the others, and is used to set the frequency appropriately. The numerical values in the figure show the total task utilizations computed using the information available at each point.

The algorithm itself (Figure 4.4) is simple and works as follows. Suppose a task T_i completes its current invocation after using cc_i cycles which are usually much smaller than its worst-case computation time C_i. Since task T_i uses no more than cc_i cycles in its current invocation, we treat the task as if its worst-case computation bound were cc_i. With the reduced utilization specified for this task, we can now potentially find a smaller scaling factor α (i.e., lower operating frequency) for which the task set remains schedulable. Trivially, given that the task set prior to this change was schedulable, the EDF schedulability test will continue to hold, and T_i (which has completed execution) will not violate its lowered maximum computing bound for the remaining time until its deadline. Therefore, the task set continues to meet both conditions C1 and C2 imposed by the real-time scheduler to guarantee timely execution, and as a result, deadline guarantees provided by EDF scheduling will continue to hold *at least* until T_i is released for its next invocation.

select_frequency():

use lowest freq. $f_i \in \{f_1, \ldots, f_m | f_1 < \cdots < f_m\}$

such that $U_1 + \cdots + U_n \leq f_i / f_m$

upon task_release(T_i):

set U_i to C_i / P_i;

select_frequency();

upon task_completion(T_i):

set U_i to cc_i / P_i;

/* cc_i is the actual cycles used this invocation */

select_frequency();

Figure 4.4: Cycle-conserving DVS for EDF schedulers

At this point, we must restore its computation bound to C_i to ensure that it will not violate the temporarily-lowered bound and compromise the deadline guarantees. At this time, it may be necessary to increase the operating frequency. At first glance, this algorithm does not appear to significantly reduce frequencies, voltages, and energy expenditure. However, since multiple tasks may be simultaneously in the reduced-utilization state, the total savings can be significant.

We could use the same schedulability-test-based approach to designing a cycle-conserving DVS algorithm for RM scheduling, but as the RM schedulability test is significantly more complex ($O(n^2)$, where n is the number of tasks to be scheduled), we will take a different approach here. We observe that even assuming tasks always require their worst-case computation times, the statically-scaled RM mechanism discussed earlier can maintain real-time deadline guarantees. We assert that as long as equal or better progress for all tasks is made here than in the worst case under the statically-scaled RM algorithm, deadlines can be met here as well, regardless of the actual operating frequencies. We will also try to avoid getting ahead of the worst-case execution pattern; this way, any reduction in the execution

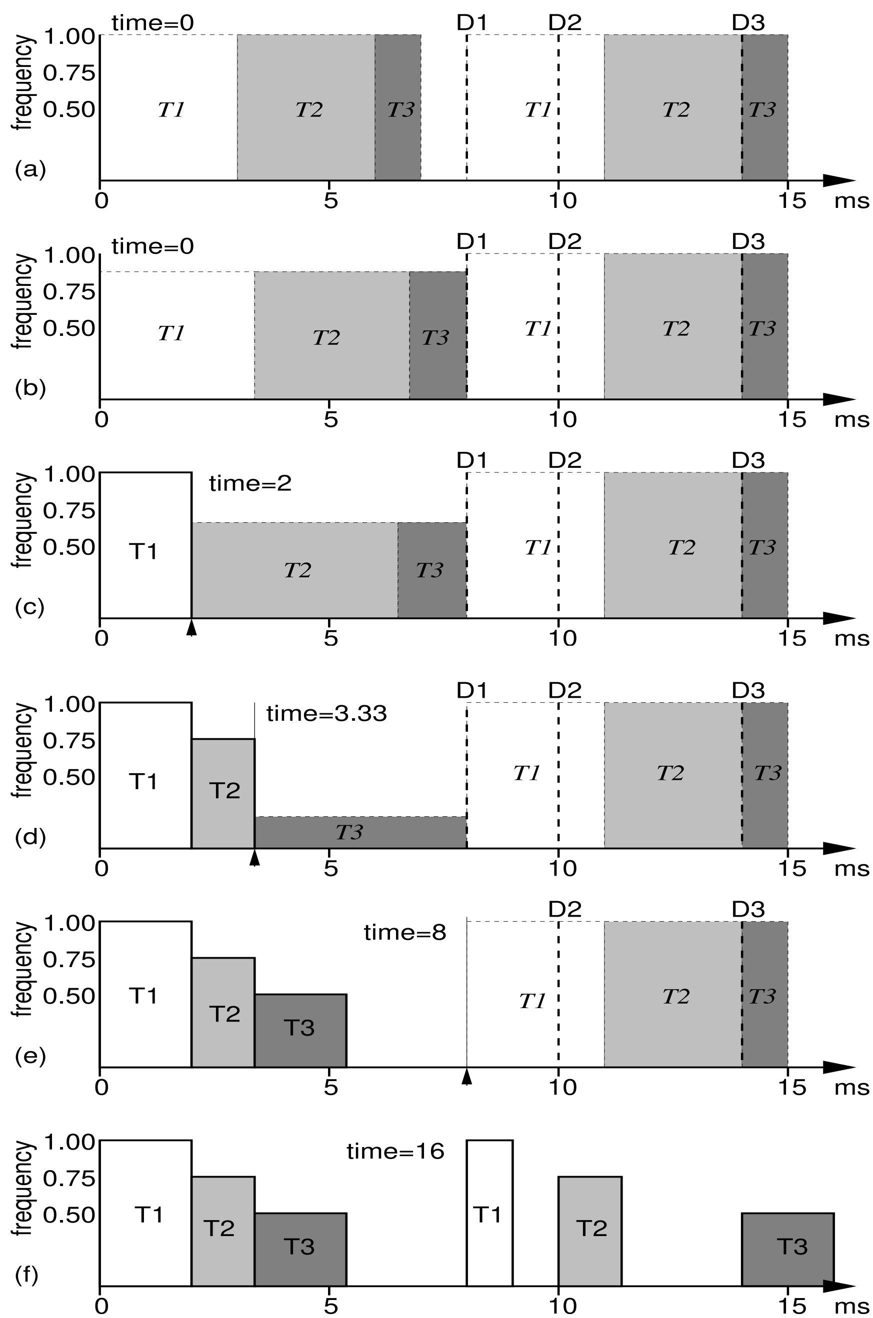


Figure 4.5: Example of cycle-conserving RM: (a) Initially use statically-scaled, worst-case RM schedule as target; (b) Determine minimum frequency so as to complete the same work by D1; rounding up to the closest discrete setting requires frequency 1.0; (c) After T1 completes (early), recompute the required frequency as 0.75; (d) Once T2 completes, a very low frequency (0.5) suffices to complete the remaining work by D1; (e) T1 is re-released, and now, try to match the work that should be done by D2; (f) Execution trace through time 16 ms.

cycles used by the tasks can be applied to reducing operating frequency and voltage. Using the same example as before, Figure 4.5 illustrates how this can be accomplished. We initially start with worst-case schedule based on static-scaling (a), which for this example, uses the maximum CPU frequency. To keep things simple, we do not look beyond the next deadline in the system. We then try to spread out the work that should be accomplished before this deadline over the entire interval from the current time to the deadline (b). This provides a minimum operating frequency value, but since the frequency settings are discrete, we round up to the closest available setting, frequency=1.0. After executing T_1, we repeat the exercise of spreading out the remaining work over the remaining time until the next deadline (c), which results in a lower operating frequency since T_1 completed earlier than its worst-case specified computing time. Repeating this at each scheduling point results in the final execution trace (f).

Although conceptually simple, the actual algorithm (Figure 4.6) for this is somewhat complex due to a number of counters that must be maintained. In this algorithm, we need to keep track of the worst-case remaining cycles of computation, c_left_i, for each task T_i. When task T_i is released, c_left_i is set to C_i. We then determine the progress that the static voltage scaling RM mechanism would make in the worst case by the earliest deadline for *any* task in the system. We obtain s_j and s_m, the number of cycles to this next deadline, assuming operation at the statically-scaled and the maximum frequencies, respectively. The s_j cycles are allocated to the tasks according to RM priority order, with each task T_i receiving an allocation $d_i \leq c\_left_i$ corresponding to the number of cycles that it would execute under the statically-scaled RM scenario over this interval. As long as we execute at least d_i cycles for each task T_i (or if T_i completes) by the next task deadline, we are keeping pace with the worst-case scenario, so we set execution speed using the sum of the d values. As tasks execute, their c_left and d values are decremented. When a task T_i completes, c_left_i and d_i are both set to 0, and the frequency and voltage are changed. Because we use this pacing criteria to select the operating frequency, this algorithm guarantees that at any task deadline, all tasks that would have completed execution in the worst-case, statically-scaled RM schedule would also have completed here, hence meeting all deadlines.

These algorithms dynamically adjust frequency and voltage, reacting to the actual computational requirements of the real-time tasks. At most, they require 2 frequency/voltage

```
assume f_j is frequency set by static scaling algorithm

select_frequency():
    set s_m = max_cycles_until_next_deadline();
    use lowest freq. f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
    such that (d_1 + ··· + d_n)/s_m ≤ f_i/f_m

upon task_release(T_i):
    set c_left_i = C_i;
    set s_m = max_cycles_until_next_deadline();
    set s_j = s_m * f_j/f_m;
    allocate_cycles (s_j);
    select_frequency();

upon task_completion(T_i):
    set c_left_i = 0;
    set d_i = 0;
    select_frequency();

during task_execution(T_i):
    decrement c_left_i and d_i;

allocate_cycles(k):
    for i = 1 to n,   T_i ∈ {T_1, ..., T_n | P_1 ≤ ··· ≤ P_n}
                      /* tasks sorted by period */
        if ( c_left_i < k )
            set d_i = c_left_i;
            set k = k - c_left_i;
        else
            set d_i = k;
            set k = 0;
```

Figure 4.6: Cycle-conserving DVS for RM schedulers

switches per task per invocation (once each at release and completion), so any overheads for hardware voltage change can be accounted in the worst-case computation time allocations of the tasks. As we will see later, the algorithms can achieve significant energy savings without affecting real-time guarantees.

4.2.5 Look-Ahead RT-DVS

The final (and most aggressive) RT-DVS algorithm that we introduce in this chapter attempts to achieve even better energy savings using a look-ahead technique to determine future computation need and defer task execution. The cycle-conserving approaches discussed above assume the worst case initially, execute tasks at a high frequency until some of them complete, and only then reduce operating frequency and voltage. In contrast, the look-ahead scheme tries to defer as much work as possible, and sets the operating frequency to complete the minimum work that must be done now to ensure all future deadlines are met. Of course, a result of this may be that we are required to run at high frequencies later in order to complete all of the deferred work in time. On the other hand, if tasks use much less than their worst-case computing time allocations, the peak execution rates for deferred work may never be needed, and this heuristic will allow the system to continue operating at a low frequency and voltage while completing all tasks by their deadlines.

Continuing with the example used earlier, we illustrate how a look-ahead RT-DVS EDF algorithm works in Figure 4.7. The goal is to defer work beyond the earliest deadline in the system (D_1) so that we can operate at a low frequency between the current time and D_1. We allocate time in the schedule for the worst-case execution of each task, starting with the task with the latest deadline, T_3. We spread out T_3's work between D_1 and its own deadline, D_3, subject to a constraint reserving capacity for future invocations of the other tasks (a). After allocating T_3 and reserving capacity for future invocations of T_1, we repeat this step for T_2, which cannot entirely fit between D_1 and D_2. Additional work for T_2 and all of T_1 are allotted before D_1 (b). We note that more of T_2 could be deferred beyond D_1 if we moved all of T_3 after D_2, but for simplicity, this is not considered. We use the work allocated before D_1 to determine the operating frequency. Once T_1 has completed, using less than its specified worst-case execution cycles, we repeat this and find a lower operating frequency (c). Continuing this method of trying to defer work beyond the next deadline in

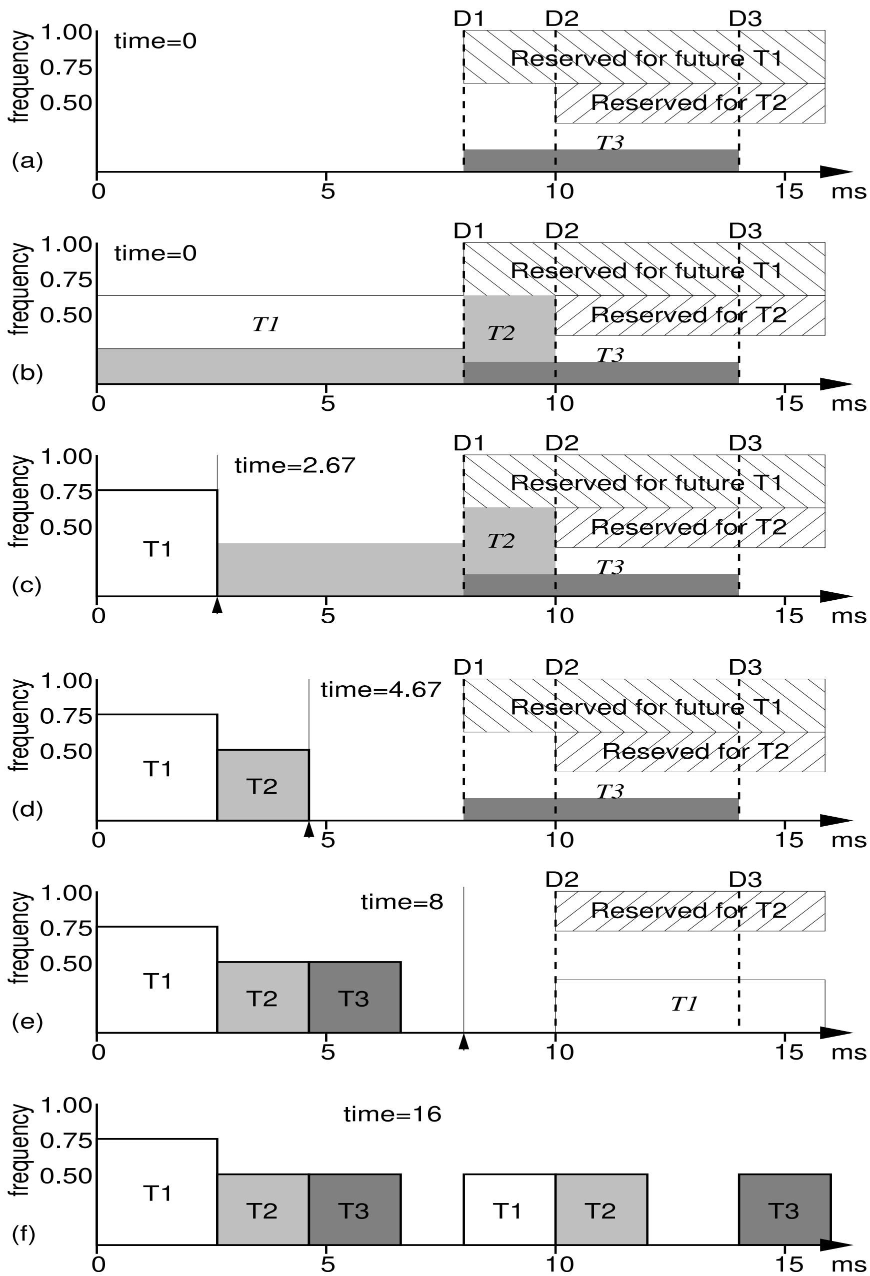


Figure 4.7: Example of look-ahead EDF: (a) At time 0, plan to defer T3's execution until after D1 (but by its deadline D3, and likewise, try to fit T2 between D1 and D2; (b) T1 and the portion of T2 that did not fit must execute before D1, requiring use of frequency 0.75; (c) After T1 completes, repeat calculations to find the new frequency setting, 0.5; (d) Repeating the calculation after T2 completes indicates that we do not need to execute anything by D1, but EDF is work-conserving, so T3 executes at the minimum frequency; (e) This occurs again when T1's next invocation is released; (f) Execution trace through time 16 ms.

```
select_frequency(x):
    use lowest freq. f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
    such that x ≤ f_i/f_m

upon task_release(T_i):
    set c_left_i = C_i;
    defer();

upon task_completion(T_i):
    set c_left_i = 0;
    defer();

during task_execution(T_i):
    decrement c_left_i;

defer():
    set U = C_1/P_1 + ··· + C_n/P_n;
    set s = 0;
    for i = 1 to n,    T_i ∈ {T_1, ..., T_n | D_1 ≥ ··· ≥ D_n}
                 /* Note: reverse EDF order of tasks */
        set U = U − C_i/P_i;
        set x = max(0, c_left_i − (1 − U)(D_i − D_n));
        set U = U + (c_left_i − x)/(D_i − D_n);
        set s = s + x;
    select_frequency (s/(D_n− current_time));
```

Figure 4.8: Look-Ahead DVS for EDF schedulers

the system ultimately results in the execution trace shown in (f).

The actual algorithm for look-ahead RT-DVS with EDF scheduling is shown in Figure 4.8. As in the cycle-conserving RT-DVS algorithm for RM, we keep track of the worst-case remaining computation c_left_i for the current invocation of task T_i. This is set to C_i on task release, decremented as the task executes, and set to 0 on completion. The major step in this algorithm is the deferral function. Here, we look at the interval until the next task deadline, try to push as much work as we can beyond the deadline, and compute the minimum number of cycles, s, that we must execute during this interval in order to meet all future deadlines. The operating frequency is set just fast enough to execute s cycles over this interval. To calculate s, we look at the tasks in reverse EDF order (i.e., latest deadline first). Assuming worst-case utilization by tasks with earlier deadlines (effectively reserving time for their future invocations), we calculate the minimum number of cycles, x, that the task must execute before the closest deadline, D_n, in order for it to complete by its own deadline. A cumulative utilization U is adjusted to reflect the actual utilization of the task for the time after D_n. This calculation is repeated for all of the tasks, using assumed worst-case utilization values for earlier-deadline tasks and the computed values for the later-deadline ones. s is simply the sum of the x values calculated for all of the tasks, and therefore reflects the total number of cycles that must execute by D_n in order for all tasks to meet their deadlines. Although this algorithm very aggressively reduces processor frequency and voltage, it ensures that there are sufficient cycles available for each task to meet its deadline after reserving worst-case requirements for higher-priority (earlier deadline) tasks.

4.2.6 Summary of RT-DVS algorithms

All of the RT-DVS algorithms we presented thus far should be fairly easy to incorporate into a real-time operating system, and do not require significant processing costs. The dynamic schemes all require $O(n)$ computation (assuming the scheduler provides an EDF sorted task list), and should not require significant processing beyond what is needed for the scheduler. The most significant overheads may come from the hardware voltage switching times. However, in all of our algorithms, no more than two switches can occur per task per invocation period, so these overheads can easily be accounted for, and added to, the

RT-DVS method	energy used
none (plain EDF)	1.0
statically-scaled RM	1.0
statically-scaled EDF	0.64
cycle-conserving EDF	0.52
cycle-conserving RM	0.71
look-ahead EDF	0.44

Table 4.4: Normalized energy consumption for the example traces

worst-case task computation times.

To conclude our series of examples, Table 4.4 shows the normalized energy dissipated in the example task (Table 4.2) set for the first 16 ms, using the actual execution times from Table 4.3. We assume that the 0.5, 0.75 and 1.0 frequency settings need 3, 4, and 5 volts, respectively, and that idle cycles consume no energy. More general evaluation of our algorithms will be done in the next section.

4.3 Simulations

We have developed a simulator to evaluate the potential energy savings from voltage scaling in a real-time scheduled system. The following subsection describes our simulator and the assumptions made in its design. Later, we show some simulation results and provide insight into the most significant system parameters affecting RT-DVS energy savings.

4.3.1 Simulation Methodology

Using C++, we developed a simulator for the operation of hardware capable of voltage and frequency scaling with real-time scheduling. The simulator takes as input a task set, specified with the period and computation requirements of each task, as well as several system parameters, and provides the energy consumption of the system for each of the algorithms we have developed. EDF and RM schedulers without any DVS support are also simulated for comparison.[3] Parameters supplied to the simulator include the machine spec-

[3]Without DVS, energy consumption is the same for both EDF and RM, so EDF numbers alone would suffice. However, since some task sets are schedulable under EDF, but not under RM, we simulate both to verify that all task sets that are schedulable under RM are also schedulable when using the RM-based RT-DVS mechanisms.

ification (a list of the frequencies and corresponding voltages available on the simulated platform) and a specification for the actual fraction of the worst-case execution cycles that the tasks will require for each invocation. This latter parameter can be a constant (e.g., 0.9 indicates that each task will use 90% of its specified worst-case computation cycles during each invocation), or can be a random function (e.g., uniformly-distributed random multiplier for each invocation).

The simulation assumes that a constant amount of energy is required for each cycle of operation at a given voltage. This quantum is scaled by the square of the operating voltage, consistent with energy dissipation in CMOS circuits ($E \propto V^2$). Only the energy consumed by the processor is computed, and variations due to different types of instructions executed are not taken into account. With this simplification, the task execution modeling can be reduced to counting cycles of execution, and execution traces are not needed. The software-controlled halt feature, available on some processors and used for reducing energy expenditure during idle, is simulated by specifying an idle level parameter. This value gives the ratio between energy consumed during a cycle while halted and that during a cycle of normal operation (e.g., a value of 0.5 indicates a cycle spent idling dissipates one half the energy of a cycle of computation). For simplicity, only task execution and idle (halt) cycles are considered. In particular, this does not consider preemption and task switch overheads or the time required to switch operating frequency or voltages. There is no loss of generality from these simplifications. The preemption and task switch overheads are the same with or without DVS, so they have no effect on relative power dissipation. The voltage switching overheads incur a time penalty, which may affect the schedulability of some task sets, but incur almost no energy costs, as the processor does not operate during the switching interval.

The real-time task sets are specified using a pair of numbers for each task, indicating its period and worst-case computation time. The task sets are generated randomly as follows. Each task has an equal probability of having a short (1–10ms), medium (10–100ms), or long (100–1000ms) period. Within each range, task periods are uniformly distributed. This simulates the varied mix of short and long period tasks commonly found in real-time systems. The computation requirements of the tasks are assigned randomly using a similar 3 range uniform distribution. Finally, the task computation requirements are scaled by a constant chosen such that the sum of the utilizations of the tasks in the task set reaches a desired value. This method of generating real-time task sets has been used previously in the development and evaluation of a real-time embedded microkernel [95]. Averaged across hundreds of distinct task sets generated for several different total worst-case utilization values, the simulations provide a relationship of energy consumption to worst-case utilization of a task set.

4.3.2 Simulation Results

We have performed extensive simulations of the RT-DVS algorithms to determine the most important and interesting system parameters that affect energy consumption. Unless specified otherwise, we assume a DVS-capable platform that provides 3 relative operating frequencies (0.5, 0.75, and 1.0) and corresponding voltages (3, 4, and 5, respectively).

In the following simulations, we compare our RT-DVS algorithms to each other and to a non-DVS system. We also include a theoretical lower bound for energy dissipation. This lower bound reflects execution throughput only, and does not consider any timing issues (e.g., whether any task is active or not). It is computed by taking the total number of task computation cycles in the simulation, and determining the absolute minimum energy with which these can be executed over the simulation time duration with the given platform frequency and voltage specification. No real algorithms can do better than this theoretical lower bound, since id does not even consider task release times or deadlines, but it is interesting to see how close our mechanisms approach this bound.

Number of tasks: In our first set of simulations, we determine the effects of varying the number of tasks in the task sets. Figure 4.9 shows the energy consumption for task sets with 5, 10, and 15 tasks for all of our RT-DVS algorithms as well as unmodified EDF. All of these simulations assume that the processor provides a perfect software-controlled halt function (so idling the processor will consume no energy), thus showing scheduling without any energy conserving features in the most favorable light. In addition, we assume that tasks do consume their worst-case computation requirements during each invocation. With these extremes, there is no difference between the statically-scaled and cycle-conserving EDF algorithms.

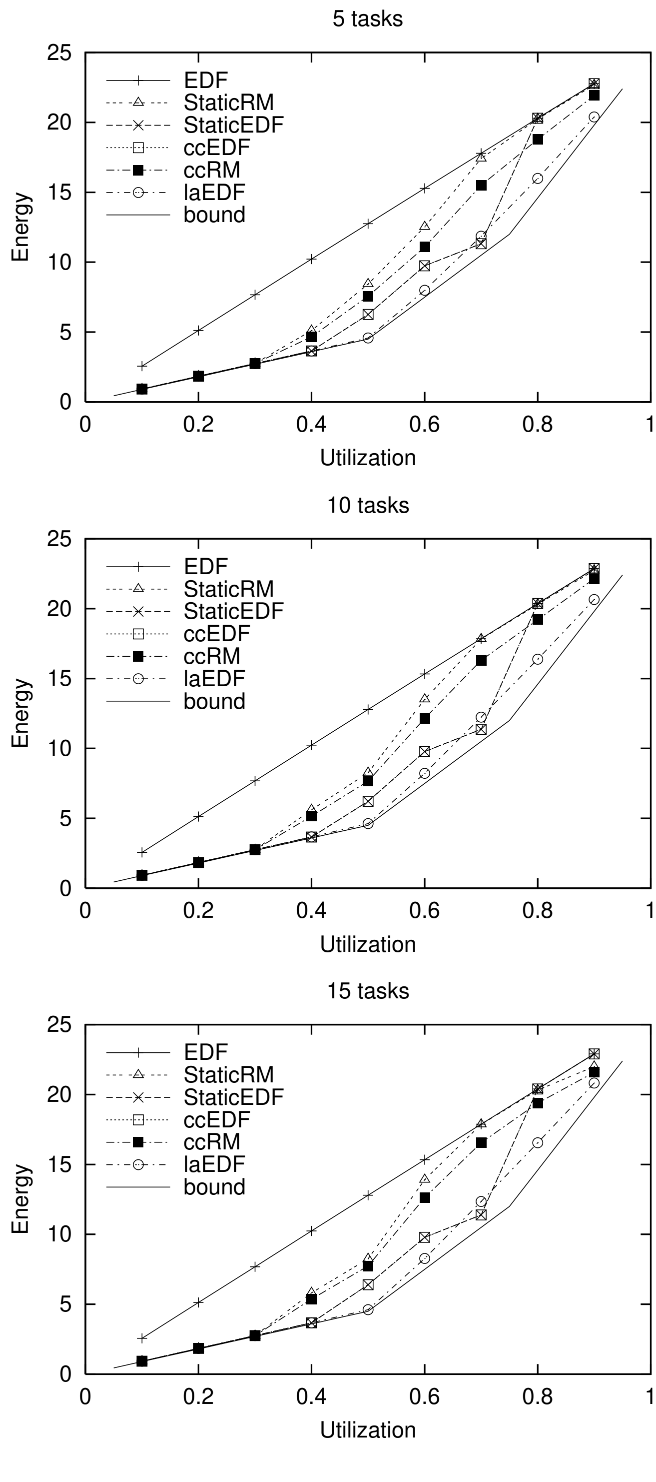


Figure 4.9: Energy consumption with 5, 10, and 15 tasks

We notice immediately that the RT-DVS algorithms show potential for large energy savings, particularly for task sets with mid-range worst-case processor utilization values. The look-ahead RT-DVS mechanism, in particular, seems able to follow the theoretical lower bound very closely. Although the total utilization greatly affects energy consumption, the number of tasks has very little effect. Neither the relative nor the absolute positions of the curves for the different algorithms shift significantly when the number of tasks is varied. Since varying the number of tasks has little effect, for all further simulations, we will use a single value.

Varying idle level: The preceding simulations assumed that a perfect software-controlled halt feature is provided by the processor, so idle time consumes no energy. To see how an imperfect halt feature affects power consumption, we performed several simulations varying the idle level factor, which is the ratio of energy consumed in a cycle while the processor is halted, to the energy consumed in a normal execution cycle. Figure 4.10 shows the results for idle level factors 0.01, 0.1, and 1.0. Since the absolute energy consumed will obviously increase as the idle state energy consumption increases, it is more insightful to look at the *relative* energy consumption by plotting the values normalized with respect to the unmodified EDF energy consumption.

The most significant result is that even with a perfect halt feature (i.e., idle level is 0), where the non-energy conserving schedulers are shown in the best light, there is still a very large percentage improvement with the RT-DVS algorithms. Obviously, as the idle level increases to 1 (same energy consumption as in normal operation), the percentage savings with voltage scaling improves. The relative performance among the energy-aware schedulers is not significantly affected by changing the idle power consumption level, although the dynamic algorithms benefit more than the statically-scaled ones. This is evident as the cycle-conserving EDF mechanism results diverge from the statically-scaled EDF results. This is easily explained by the fact that the dynamic algorithms switch to the lowest frequency and voltage during idle, while the static ones do not; with perfect idle, this makes no difference, but as idle cycle energy consumption approaches that of execution cycles, the dynamic algorithms perform relatively better. For the remainder of the simulations, we assume an idle level of 0.

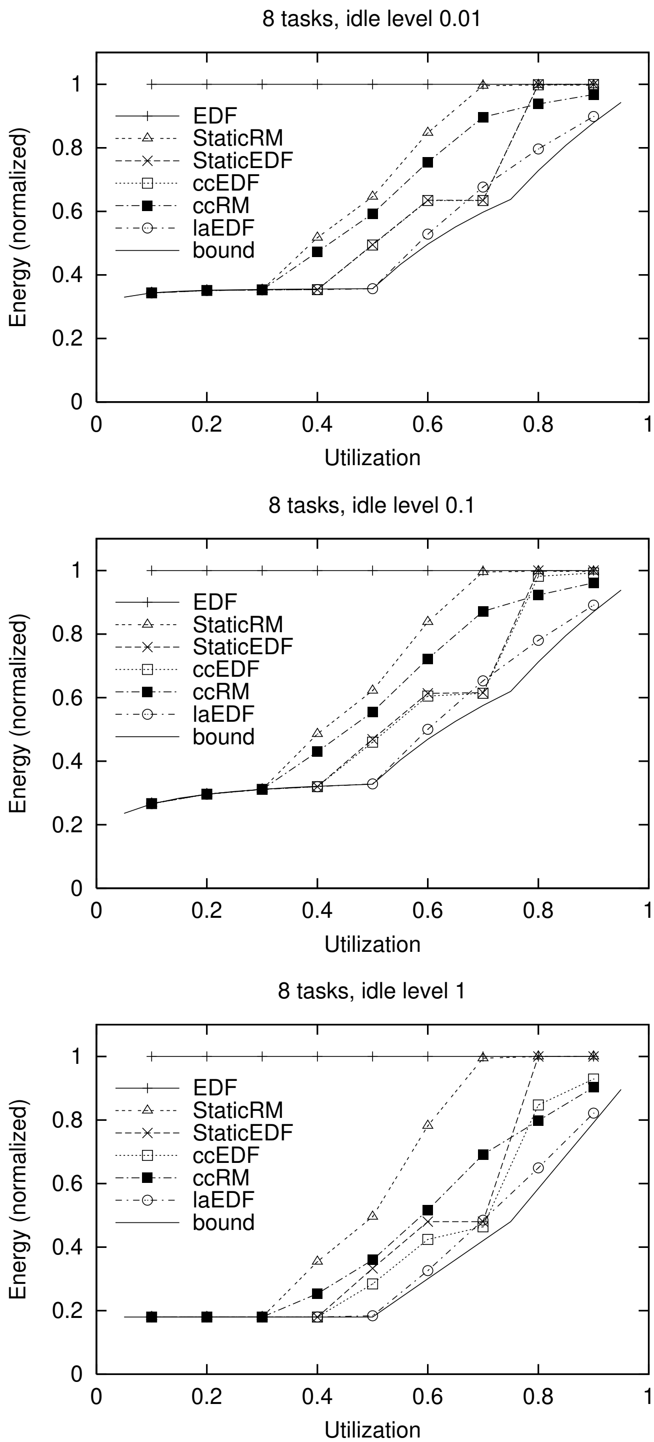


Figure 4.10: Normalized energy consumption with idle level factors 0.01, 0.1, and 1.0

Varying machine specifications: All of the previous simulations used only one set of available frequency and voltage scaling settings. We now investigate the effects of varying the simulated machine specifications. The following summarizes the hardware voltage and frequency settings, where each tuple consists of the relative frequency and the corresponding processor voltage:

machine 0: { (0.5, 3), (0.75, 4), (1.0, 5) }
machine 1: { (0.5, 3), (0.75, 4), (0.83, 4.5), (1.0, 5) }
machine 2: { (0.36, 1.4), (0.55, 1.5), (0.64, 1.6),
(0.73, 1.7), (0.82, 1.8), (0.91, 1.9), (1.0, 2.0) }

Figure 4.11 shows the simulation results for machines 0, 1, and 2. Machine 0, used in all of the previous simulations, has frequency settings that can be expected on a standard PC motherboard, although the corresponding voltage levels were arbitrarily selected. Machine 1 differs from this in that it has the additional frequency setting, 0.83. With this small change, we expect only slight differences in the simulation results with these specifications. The most significant change is seen with cycle-conserving EDF (and statically-scaled EDF, since the two are identical here). With the extra operating point in the region near the cross-over point between ccEDF and ccRM, the ccEDF algorithm benefits, shifting the cross-over point closer to full utilization.

Machine 2 is very different from the other two, and reflects the settings that may be available on a platform incorporating an AMD K6 processor with AMD's PowerNow! mechanism [2]. Again, the voltage levels are only speculated here. As it has many more settings to select from, the plotted curves tend to be smoother. Also, since the relative voltage range is smaller with this specification, the maximum savings is not as good as with the other two machine specifications. More significant is the fact that the cycle-conserving EDF outperforms the look-ahead EDF algorithm. ccEDF and staticEDF benefit from the large number of settings, since this allows them to more closely match the task set and reduce energy expenditure. In fact, they very closely approximate the theoretical lower bound over the entire range of utilizations. On the other hand, laEDF sets the frequency trying to defer processing (which, in the worst case, would require running at full speed later). With more settings, the low frequency setting is closely matched, requiring high-voltage, high-frequency processing later, hurting performance. With fewer settings, the frequency

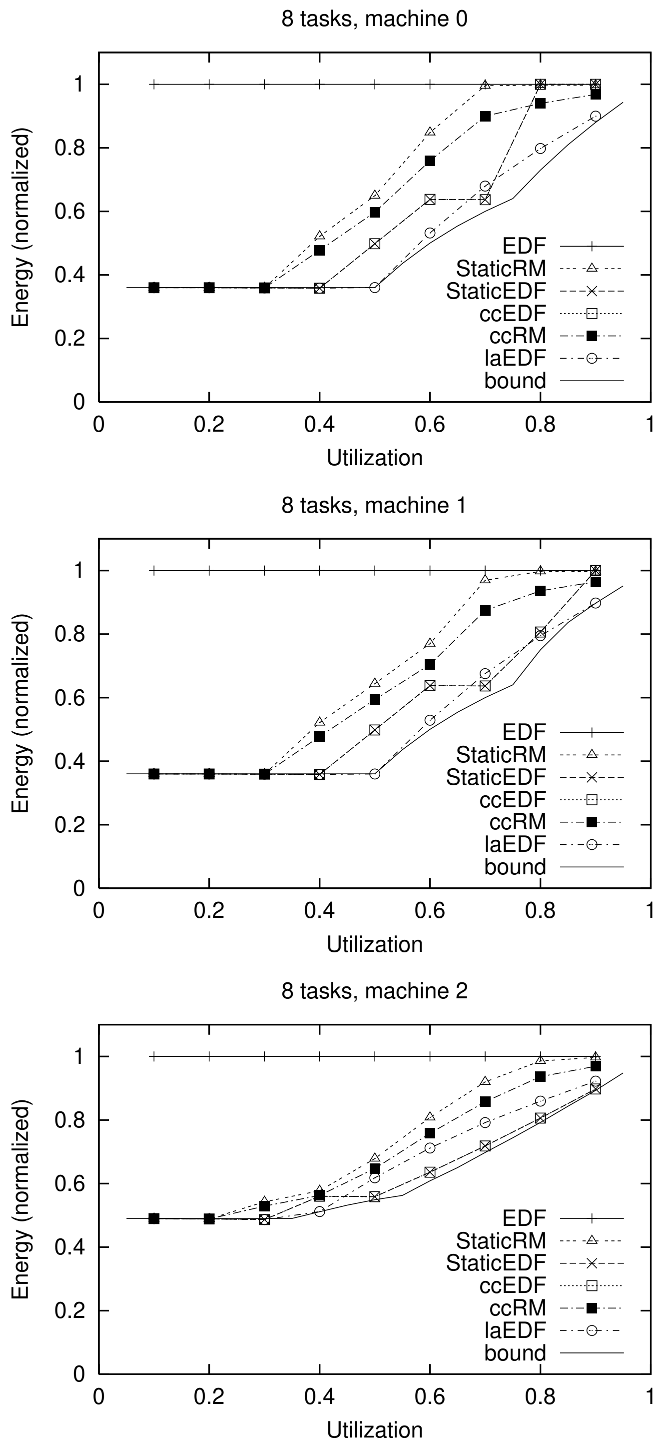


Figure 4.11: Normalized energy consumption with machine 0, 1, and 2

selected would be somewhat higher, so less processing is deferred, lessening the likelihood of needing higher voltage and frequency settings later, thus improving performance. In a sense, the error due to a limited number of frequency steps is detrimental in the ccEDF scheme, but beneficial with laEDF. These results, therefore, indicate that the energy savings from the various RT-DVS algorithms depend greatly on the available voltage and frequency settings of the platform.

Varying computation time: In this set of experiments, we vary the distribution of the actual computation required by the tasks during each invocation to see how well the RT-DVS mechanisms take advantage of task sets that do not consume their worst-case computation times. In the preceding simulations, we assumed that the tasks always require their worst-case computation allocation. Figure 4.12 shows simulation results for tasks that require a constant 90%, 70%, and 50% of their worst-case execution cycles for each invocation. We observe that the statically-scaled mechanisms are not affected, since they scale voltage and frequency based solely on the worst-case computation times specified for the tasks. The results for the cycle-conserving RM algorithm do not show significant change, indicating that it does not do a very good job of adapting to tasks that use less than their specified worst-case computation times. On the other hand, both the cycle-conserving and look-ahead EDF schemes show great reductions in relative energy consumption as the actual computation performed decreases.

Figure 4.13 shows the simulation results using tasks with a uniform distribution between 0 and their worst-case computation. Despite the randomness introduced, the results appear identical to setting computation to a constant one half of the specified value for each invocation of a task. This makes sense, since the average execution with the uniform distribution is 0.5 times the worst-case for each task. From this, it seems that the actual distribution of computation per invocation is not the critical factor for energy conservation performance. Instead, for the dynamic mechanisms, it is the average utilization that determines relative energy consumption, while for the static scaling methods, the worst-case utilization is the determining factor. The exception is the ccRM algorithm, which, albeit dynamic, has results that primarily reflect the worst-case utilization of the task set.

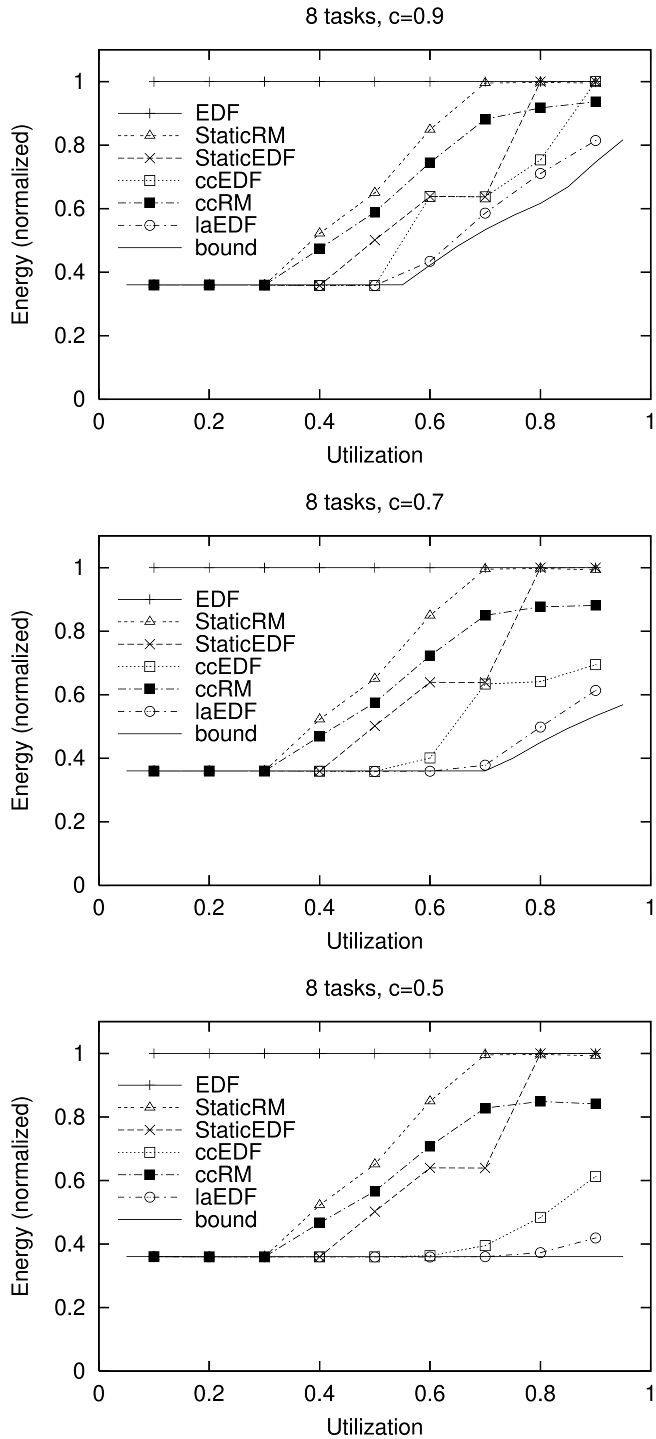


Figure 4.12: Normalized energy consumption with computation set to fixed fraction of worst-case allocation

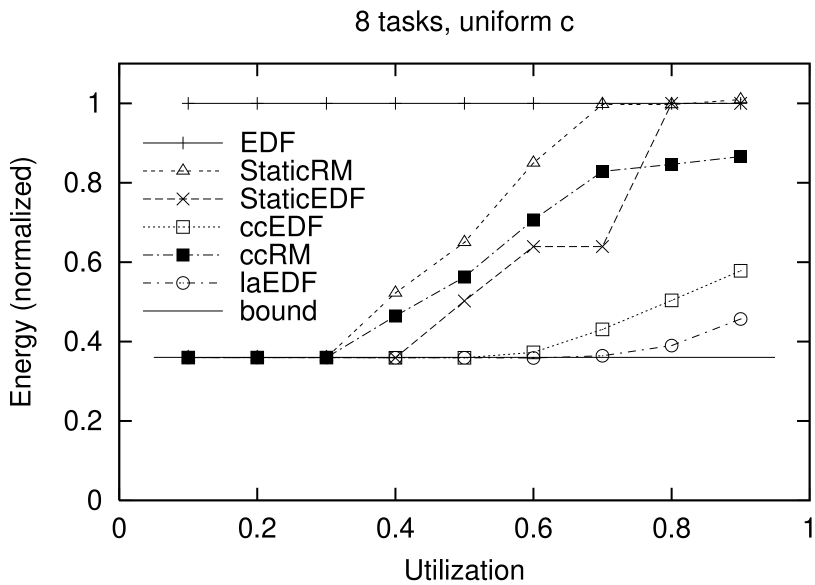


Figure 4.13: Normalized energy consumption with uniform distribution for computation

4.4 Implementation

This section describes our implementation of a real-time scheduled system incorporating the proposed DVS algorithms. We will discuss the architecture of the system and present measurements of the actual energy savings with our RT-DVS algorithms.

4.4.1 Hardware Platform

Although we developed the RT-DVS mechanisms primarily for embedded real-time devices, our prototype system is implemented on the PC architecture. The platform is a Hewlett-Packard N3350 notebook computer, which has an AMD K6-2+ [2] processor with a maximum operating frequency of 550 MHz. Some of the power consumption numbers measured on this laptop were shown earlier in Table 4.1. This processor features *PowerNow!*, AMD's extensions that allow the processor's clock frequency and voltage to be adjusted dynamically under software control. We have also looked into a similar offering from Intel called *SpeedStep* [29], but this controls voltage and frequency through hardware external to the processor. Although it is possible to adjust the settings under software control, we were not able to determine the proper output sequences needed to control the external hardware. We do not have experience with the Transmeta Crusoe processor [87] or with various embedded processors (such as Intel XScale [28]) that are now supporting DVS.

The K6-2+ processor specification allows system software to select one of eight different frequencies from its built-in PLL clock generator (200 to 600 MHz in 50 MHz increments, skipping 250), limited by the maximum processor clock rate (550 MHz here). The processor has 5 control pins that can be used to set the voltage through an external regulator. Although 32 settings are possible, there is only one that is explicitly specified (the default 2.0V setting); the rest are left up to the individual manufacturers. HP chose to incorporate only 2 voltage settings: 1.4V and 2.0V. The voltage and frequency selectors are independently set, so we need a function to map each frequency to the appropriate available voltage level. As there are no such specifications publicly available, we determined this experimentally. The processor was stable up to 450 MHz at 1.4 V, and needed the 2.0 V setting for higher frequencies. Stability was checked using a set of CPU-intensive benchmarks (`mpg123` in a loop and Linux kernel compile), and verifying proper behavior. We note that this empirical study used a sample size of one, so the actual frequency to voltage mappings may vary for other machines, even of the same model.

The processor has a mandatory stop interval associated with every change of the voltage or frequency transition, during which the processor halts execution. This mandatory halt duration is meant to ensure that the voltage supply and clock have time to stabilize before the processor continues execution. As the characteristics of different hardware implementations of the external voltage regulators can vary greatly, this stop duration is programmable in multiples of 41 μs (4096 cycles of the 100 MHz system bus clock). According to our experience, it takes negligible time for frequency changes to occur. Using the CPU timestamp register (basically a cycle counter), which continues to increment during the halt duration, we observed that around 8200 cycles occur during any transition to 200 MHz, and around 22500 cycles for a transition to 550 MHz, both with the minimum interval of 41 μs. This indicates that the frequency of the CPU clock changes quickly and that most of the halt time is spent at the target frequency. We do not know the actual time required for voltage transition to occur, but our experiments using a halt duration value of 10 (approximately 0.4 ms) resulted in no observable instability. The switching overheads in our system, therefore, are 0.4 ms when voltage changes, and 41 μs when only frequency

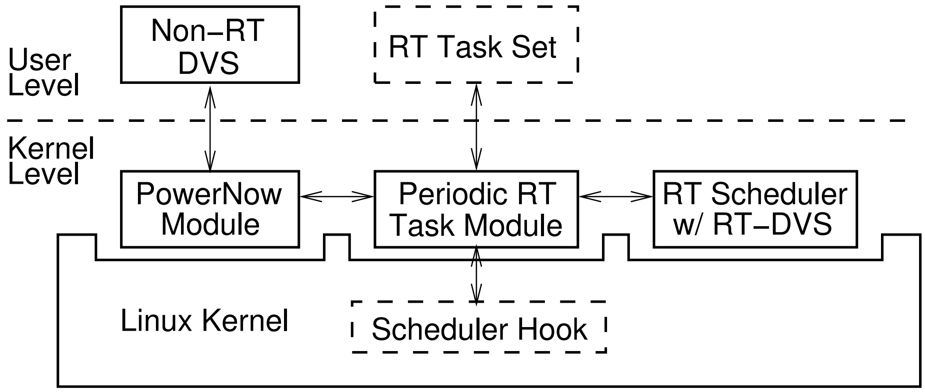


Figure 4.14: Software architecture for RT-DVS implementation

changes. As mentioned earlier, we can account for this switching overhead in the computation requirements of the tasks, since at most only two transitions are attributable to each task in each invocation.

4.4.2 Software Architecture

We implemented our algorithms as extension modules to the Linux 2.2.16 kernel. Although it is not a real-time operating system, Linux is easily extended through modules and provides a robust development environment familiar to us. The high-level view of our software architecture is shown in Figure 4.14. The approach taken for this implementation was to maximize flexibility and ease of use, rather than optimize for performance. As such, this implementation serves as a good proof-of-concept, rather than the ideal model. By implementing our kernel-level code as Linux kernel modules, we avoided any code changes to the Linux kernel, and these modules should be able to plug into unmodified 2.2.x kernels.

The central module in our implementation provides support for periodic real-time tasks in Linux. This is done by attaching call-back functions to hooks inside the Linux scheduler and timer tick handlers. This mechanism allows our modules to provide tight timing control as well as override the default Unix scheduling policy for our real-time tasks. Note that this module does not actually define the real-time scheduling policy or the DVS algorithm. Instead, we use separate modules that provide the real-time scheduling policy and the RT-DVS algorithms. One such RT scheduler/DVS module can be loaded on the system at a time. By separating the underlying periodic RT support from the scheduling and DVS policies, this architecture allows dynamic switching of these latter policies without shutting down the system or the running RT tasks. (Of course, during the switch-over time between these policy modules, a real-time scheduler is not defined, and the timeliness constraints of any running RT tasks may not be met). The last kernel module in our implementation handles the access to the PowerNow! mechanism to adjust clock speed and voltage. This provides a clean, high-level interface for setting the appropriate bits of the processor's special feature register for any desired frequency and voltage level.

The modules provide an interface to user-level programs through the Linux `/procfs` filesystem. Tasks can use ordinary file read and write operations to interact with our modules. In particular, a task can write its required period and maximum computing bound to our module, and it will be made into a periodic real-time task that will be released periodically, scheduled according to the currently-loaded policy module, and will receive static priority over non-RT tasks on the system. The task also uses writes to indicate the completion of each invocation, at which time it will be blocked until the next release time. As long as the task keeps the file handle open, it will be registered as a real-time task with our kernel extensions. Although this high-level, filesystem interface is not as efficient as direct system calls, it is convenient in this prototype implementation, since we can simply use `cat` to read from our modules and obtain status information in a human readable form. The PowerNow! module also provides a `/procfs` interface. This will allow for a user-level, non-RT DVS demon, implementing algorithms found in other DVS literature, or to manually deal with operating frequency and voltage through simple shell commands.

4.4.3 Measurements and Observations

We performed several experiments with our RT-DVS implementation and measured the actual energy consumption of the system. Figure 4.15 shows the setup used to measure energy consumption. The laptop battery is removed and the system is run using the external DC power adapter. Using a special current probe, a digital oscilloscope measures the power consumed by the laptop as the product of the current and voltage supplied. This basic methodology is very similar to the one used in the PowerScope [22], but instead of a slow digital multimeter, we use an oscilloscope that can show the transient behavior and also provide the true average power consumption over long intervals. Using the long duration

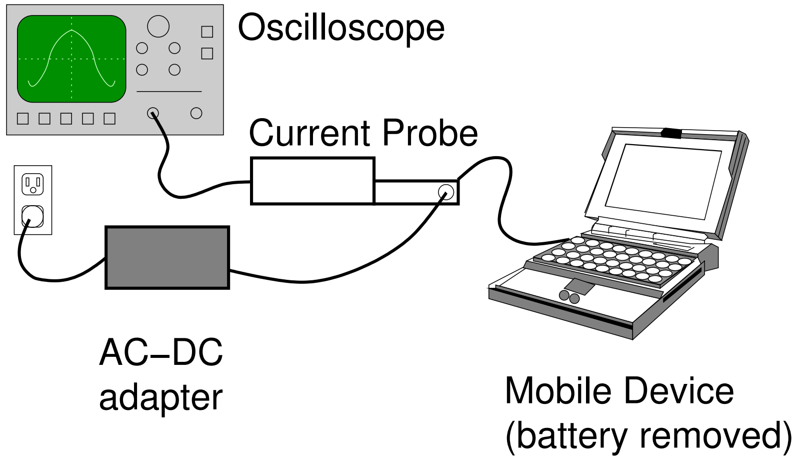


Figure 4.15: Power measurement on laptop implementation

acquisition capability of the digital oscilloscope, our power measurements are averaged over 15 to 30 second intervals.

Figure 4.16 shows the actual power consumption measured for our RT-DVS algorithms while varying worst-case CPU utilization for a set of 5 tasks which always consume 90% of their worst-case computation allocated for each invocation. The measurements reflect the total system power, not just the CPU energy dissipation. As a result, there is a constant, irreducible power drain from the system board consumption (the display backlighting was turned off for these measurements; with this on, there would have been an additional constant 6 W to each measurement). Even with this overhead, our RT-DVS mechanisms show a significant 20% to 40% reduction in power consumption, while still providing the deadline guarantees of a real-time system.

Figure 4.17 shows a simulation with identical parameters (including the 2 voltage-level machine specification) to these measurements. The simulation only reflects the processor's energy consumption, so does not include any energy overheads from the rest of the system. It is clear that, except for the addition of constant overheads in the actual measurements, the results are nearly identical. This validates our simulation results, showing that the results we have seen earlier really hold in real systems, despite the simplifying assumptions in the simulator. The simulations are accurate and may be useful for predicting the performance of RT-DVS implementations.

We also note two interesting phenomena that should be considered when implement-

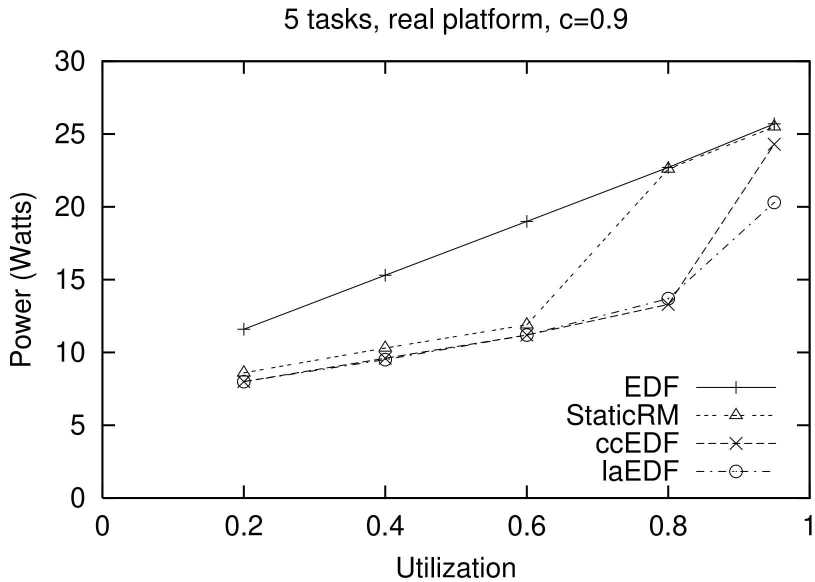


Figure 4.16: Power consumption on actual platform

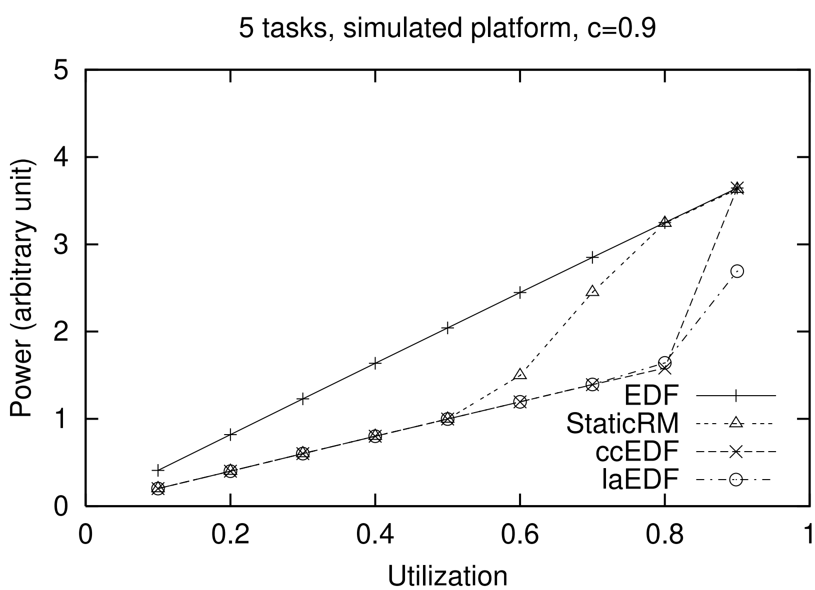


Figure 4.17: Power consumption on simulated platform

ing a system with RT-DVS. First, we noticed that the very first invocation of a task may overrun its specified computing time bound. This occurs only on the first invocation, and is caused by "cold" processor and operating system state. In particular, when the task begins execution, many cache misses, translation-look-aside buffer (TLB) misses, and page faults occur in the system (the last may be due to the copy-on-write page allocation mechanism used by Linux). These processing overheads count against the task's execution time, and may cause it to exceed its bound. On subsequent invocations, the state is "warm," and this problem disappears. This is due to the large difference between worst-case and average-case performance on general-purpose platforms, and explains why real-time systems tend to use specialized platforms to decrease or eliminate such variations in performance.

The second important observation is that the dynamic addition of a task to the task set may cause transient missed deadlines unless one is very careful. Particularly with the more aggressive RT-DVS schemes, the system may be so closely matched to the current task set load that there may not be sufficient processor cycles remaining before the task deadlines to also handle the new task. One solution to this problem is to immediately insert the task into task set, so DVS decisions are based on the new system characteristics, but defer the initial release of the new task until the current invocations of all existing tasks have completed. This ensures that the effects of past DVS decisions, based on the old task set, will have expired by the time the new task is actually begins execution.

4.5 Related Work

Recently, there have been a large number of publications describing DVS techniques. Most of them present algorithms that are very loosely-coupled with the underlying OS scheduling and task management systems, relying on average processor utilization to perform voltage and frequency scaling [23, 60, 90]. They basically match the operating frequency to some weighted average of the current processor load (or conversely, idle time) using a simple feedback mechanism. Although these mechanisms result in close adaptation to the workload and large energy savings, they are unsuitable for real-time systems. More recent DVS research [20, 47] have shown methods of maintaining good interactive performance for general-purpose applications with voltage and frequency scaling. This is done through prediction of episodic interaction [20] or by applying soft deadlines and estimating task work distributions [47]. These methods show good results for maintaining short response times in human-interactive and multimedia applications, but are not intended for the stricter timeliness constraints of real-time systems.

Some DVS work has produced algorithms closely tied to the scheduler [59, 62, 69], with some claiming real-time capability. These, however, take a very simplistic view of real-time tasks — only taking into account a single execution and deadline for a task. As such, they can only handle sporadic tasks that execute just once. They cannot handle the most important, canonical model of real-time systems, which uses periodic real-time tasks. Furthermore, it is not clear how new tasks entering the system can be handled in a timely manner, especially since all of the tasks are single-shot, and since the system may not have sufficient computing resources after having adapted so closely to the current task set.

When RT-DVS was originally proposed [65], there were only four papers [24, 36, 56, 82] that dealt with DVS in a true real-time system's perspective. The first paper [36] uses a combined offline and online scheduling technique. A worst-case execution time (WCET) schedule, which provides the ideal operating frequency and voltage schedule assuming that tasks require their worst-case computation time, is calculated offline. The online scheduler further reduces frequency and voltage when tasks use less than their requested computing quota, but can still provide deadline guarantees by ensuring all invocations complete no later than in the WCET schedule. This is much more complicated than the algorithms we have presented, yet cannot deal effectively with dynamic task sets.

The second, a work-in-progress paper [82], presents two mechanisms for RT-DVS. One mechanism attempts to calculate the best feasible schedule; this is a computationally-expensive process and can only be done offline. The other is a heuristic based around EDF that tests schedulability at each scheduling point. The details of this online mechanism were not presented in [82]. Moreover, the assumption of a common period for all of the tasks is somewhat unrealistic — even if a polynomial transformation is used to produce common periods, they may need to schedule over an arbitrarily long planning cycle in their algorithm.

The third paper [56] looks at DVS from the application side. It presents a mechanism by which the application monitors the progress of its own execution, compares it to the profiled worst-case execution, and adjusts the processor frequency and voltage accordingly. The compiler inserts this monitoring mechanism at various points in the application. It is not clear how to determine the locations of these points in a task/application, nor how this mechanism will scale to systems with multiple concurrent tasks/applications.

The fourth RT-DVS paper [24] combines offline analysis with online slack-time stealing [42] and dynamic, probability-based voltage scaling. Offline analysis provides minimum operating rates for each task based on worst-case execution time. This is used in conjunction with a probability distribution for actual computation time to change frequency and voltage without violating deadlines. Excess time is used to run remaining tasks at lower CPU frequencies.

These papers, and indeed almost all papers dealing with DVS, only present simulations of algorithms. In contrast, we present fairly simple, online mechanisms for RT-DVS that work within the common models, assumptions, and contexts of real-time systems. We implemented and demonstrated RT-DVS in a real, working system. A recent paper [68] also describes a working DVS implementation, using a modified StrongARM embedded system board, which is used to evaluate a DVS scheduler in [69].

Research on RT-DVS techniques continues to be an active research area, and many papers now extend prior pioneering approaches or develop, new, more aggressive mechanisms of employing DVS in real-time systems [16, 33, 70]. Techniques have also been extended to support multiprocessor real-time systems [93].

In addition to DVS, there has been much research regarding other energy-conserving issues, including work on application adaptation [21] and communication-oriented energy conservation [35]. These issues are orthogonal to DVS and complementary to our RT-DVS mechanisms.

4.6 Conclusions

In this chapter, we have presented several novel algorithms for real-time dynamic voltage scaling that, when coupled with the underlying OS task management mechanisms and real-time scheduler, can achieve significant energy savings, while simultaneously preserving timeliness guarantees made by real-time scheduling. We first presented extensive simulation results, showing the most significant parameters affecting energy conservation through RT-DVS mechanisms, and the extent to which CPU power dissipation can be reduced. In particular, we have shown that the number of tasks and the energy efficiency of idle cycles do not greatly affect the relative savings of the RT-DVS mechanisms, while the voltage and frequency settings available on the underlying hardware and the task set CPU utilizations profoundly affect the performance of our algorithms. Furthermore, our look-ahead and cycle-conserving RT-DVS mechanisms can achieve close to the theoretical lower bound on energy. We have also implemented our algorithms and, using actual measurements, have validated that significant energy savings can be realized through RT-DVS in real systems. Additionally, our simulations do predict accurately the energy consumption characteristics of real systems. Our measurements indicate that 20% to 40% energy savings can be achieved, even including irreducible system energy overheads and using task sets with high values for both worst- and average- case utilizations.

Additionally, although developed for portable devices, RT-DVS is applicable widely in general real-time systems. The energy savings works well for extending battery life in portable applications, but can also reduce the heat generated by the real-time embedded controllers in various factory or home automation products, or even reduce cooling requirements and costs in large-scale, multiprocessor supercomputers.

CHAPTER 5

Energy-aware Quality of Service (EQoS)

The preceding chapters have considered two aspects of a software-centric approach to improving the energy efficiency of operating systems: directly improving the processing time and energy overheads of services, and scheduling algorithms to exploit hardware techniques such as power-down and voltage-scaling, while maintaining strict timeliness guarantees needed in a real-time embedded platform. These techniques together can effectively extend the battery-life of a device. However, they do not address whether the limited available energy is used in the most beneficial manner, or squandered on useless tasks. A third approach to software-centric power-management, a comprehensive mechanisms of task adaptation, is needed in order to make the best use of the available energy resources in an embedded, mobile device.

This chapter develops a new framework called *Energy-aware Quality of Service* (EQoS) that can manage real-time tasks and adapt their execution to maximize the benefits of their computation for a limited energy budget. The concept of an adaptive real-time task and the notion of utility, a measure of the benefit or value gained from their execution, are introduced. Optimal algorithms and heuristics are developed to maximize the utility of the system for a desired system runtime and a given energy budget, and then extended to optimize utility without regard to runtime. We demonstrate the effects of Dynamic Voltage Scaling (DVS) on this system and how EQoS in conjunction with DVS can provide significant gains in utility for fixed energy budgets. Finally, we evaluate this framework through both simulations and experimentation on a working implementation.

5.1 Introduction

With ever-improving semiconductor and architectural technologies, microprocessors have been improving in performance at an exponential rate. This rapid improvement in performance comes at a cost, in terms of system complexity and power dissipation. Although the move to finer-width fabrication technology and lower voltage devices allows lower-power circuits, the rate of increase in complexity, speed, and size of microprocessors has resulted in increasingly power-hungry devices. In contrast, battery and energy storage technologies have been improving at a much slower pace, and as a result, are falling further behind in relation to the energy demands of newer processors.

Energy management has, therefore, become a critical issue in all portable and mobile computing platforms. In embedded systems used for control or communications, this is of even greater importance, as it is often impossible to increase energy storage capacity in such systems, and the consequence of running out of energy can be catastrophic, rather than merely inconvenient as in consumer devices.

Several approaches to energy management have been proposed and attempted. Most consumer computing platforms implement either Advanced Power Management (APM) or Advanced Configuration and Power Interface (ACPI) [1] to manage the energy consumed by the system. Although implemented very differently, both of these are used in general-purpose platforms to power down certain hardware devices, or place them in low-power modes when not used for some period of time. These techniques work well in office computers that are powered on all night, or laptops that can shut off their modems or network interface cards when not used, and suspend the session to disk when battery is low.

Although the ability to put idle devices into low-power, standby modes is undoubtedly useful, there are limits to such techniques in embedded control applications, where the system is essentially always in an active state. In such cases, we need techniques that can operate at fine granularities to conserve energy while the system and devices operate. As the processor is often the single largest consumer of energy in most small systems (e.g., PDAs and palmtops), these mechanisms tend to focus on reducing the power of a running microprocessor. Table 5.1 shows measured energy consumption on a modern laptop, illustrating the dominance of CPU on power dissipation. One very simple and effective

Screen	CPU subsystem	Disk	Power
On	Idle	Spinning	13.5 W
On	Idle	Standby	13.0 W
Off	Idle	Standby	7.1 W
Off	Peak Load	Standby	27.3 W

Table 5.1: Power dissipation of laptop (HP N3350) components.

mechanism requires a *halt* instruction on the processor that stops the execution core until some interrupt triggers resumption. When this instruction is encountered, the CPU is effectively shut off (from an energy standpoint). Using this in an idle task can conserve almost all of the energy that would otherwise be dissipated executing idle loops. Ideally, with this mechanism, only cycles spent performing useful work will consume energy on the processor.

More advanced techniques can further reduce energy consumed even for the non-idle cycles. *Dynamic Voltage Scaling* (DVS) [90] reduces the frequency of the processor until it is just fast enough to complete all useful work, eliminating idle cycles. Since the speed at which the circuits operate is directly related to the voltage applied, it is possible to reduce the voltage when the frequency is reduced. As energy dissipated per cycle varies quadratically with voltage, DVS can potentially conserve significant amounts of energy, provided the appropriate voltage- and frequency- varying hardware is available. Such DVS hardware is now incorporated in the latest microprocessors [3, 28, 29, 87]. Assuming that timing guarantees can be preserved, embedded real-time systems can particularly benefit from both CPU-halt and DVS. As these systems are designed around worst-case execution times (WCETs) of tasks, which are often much larger than the average case, a significant amount of energy would otherwise be wasted in executing idle loops.

All of these process, battery, and DVS technologies, together, can be seen as simply increasing the total number of cycles of computation a particular-sized battery-operated device can perform. However, these provide no guidelines as to how the scarce energy may be best allocated to perform useful computation. The goal of this chapter is to introduce the new concept of *Energy-aware Quality of Service* (EQoS), a framework that can be used to maximize the total value gained from performing computation in an energy-restricted environment. In particular, we formulate energy adaptation of task sets into a tractable, optimal-selection problem, and develop solutions that, in conjunction with CPU-halt and DVS mechanisms, meet system runtime goals while maximizing the value gained from the execution of tasks.

In the next section, we describe the overall EQoS framework, and then introduce various optimal algorithms and heuristics for energy and task adaptation. Following this, we describe our implementation of EQoS, and present results from detailed simulations and actual (experimental) measurements. After relating this work with existing literature, we conclude and highlight some future research directions.

5.2 Energy-Aware Quality of Service

In order to improve energy usage and maximize the benefits of computation, we introduce a comprehensive Energy-Aware Quality of Service (EQoS) framework to regulate the consumption of scarce energy resources. In particular, the EQoS framework will allow the dynamic allocation and reclamation of energy resources from the various applications running on an embedded device. The key novel aspects of the EQoS framework are that it:

1. Brings together various technologies and techniques (including some not intended for energy conservation) to make best use of limited stored energy;

2. Varies the service level provided to each task to meet system runtime goals and maximize the total value of computation; and,

3. Formulates this comprehensive energy adaptation into a tractable, optimal-selection problem.

This entails adapting tasks to the energy available and executing them at various service quality levels, which in turn incur varying power dissipation rates. The framework to implement this comprehensive adaptation consists of components spanning from the hardware level up to the application level, as summarized in Figure 5.1. The most important technology components integrated into the EQoS framework are:

1. Low-level mechanisms to reclaim energy and reduce idle-time waste.

2. Methods of executing real-time tasks at varying quality of service and energy levels.

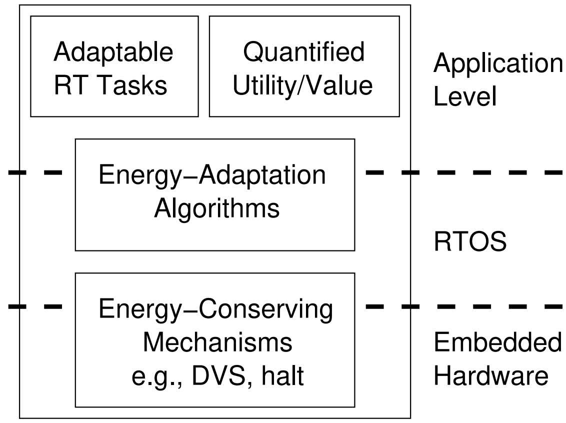


Figure 5.1: EQoS framework overview.

3. Methods of specifying task energy requirements and adaptation limits.
4. Algorithms to maximize benefits of system execution for limited energy.

The individual components of our EQoS framework are detailed below, with discussions on constraints and restrictions necessary to keep the adaptation of the real-time task set to maximize benefits of computation a tractable optimization problem.

5.2.1 Energy-Conserving Mechanisms

When employed aggressively, low-level mechanisms to conserve energy in the hardware can significantly reduce CPU power dissipation. The techniques mentioned earlier work well and are incorporated in our EQoS framework. Both the processor halt and the DVS mechanisms eliminate processor energy costs associated with idle time. DVS goes further and reduces the energy overheads of over-engineered computation capacity, by reducing per-cycle energy costs when frequency and voltage are reduced to match the actual processing load. Hence, when computational resources are not fully utilized, these mechanisms can reclaim the energy that would otherwise be wasted on idle processor cycles. One goal of EQoS adaptation is to reduce the computational load of the system to allow the low-level mechanisms to kick in, provide large energy savings, and meet system runtime requirements.

However, there are several critical problems in implementing such schemes in a real-time embedded system. In particular, varying frequency influences the execution time of the tasks in the system. As real-time systems must provide strict timeliness guarantees, changes in CPU frequency can result in timing constraint violations and deadline misses if not carefully implemented. Although many DVS mechanisms exist for general-purpose systems, very few are available for real-time systems. Recent research [24, 65] has produced some practical real-time DVS (RT-DVS) mechanisms. The EQoS framework uses RT-DVS techniques that defer task execution, reclaim unused processing cycles online, and aggressively reduce processor frequency and voltage. Assuming the task set is schedulable, the algorithms take real-time characteristics into account and ensure all deadlines are met while reducing energy expenditure.

However, the actual energy benefits of DVS algorithms depend entirely on the actual execution times of the tasks in the system, which in general are not predictable. This can cause a significant error in any adaptation that tries to achieve a system runtime requirement. Our EQoS framework deals with this problem by introducing the concept of an idealized DVS response, and using this to account for the effects of DVS. This is discussed in detail in Section 5.3.4.

5.2.2 Varying QoS for Real-Time Tasks

The low-level energy-conserving mechanisms are effective at converting any surplus computing capacity into significant energy savings. The goal of our EQoS mechanism is to maximize their effectiveness through selectively reducing load in the system by adapting the real-time task set to the available energy. When the system is energy-constrained, if we can reduce the load on the system, then RT-DVS mechanisms can kick in to extend system runtime and improve per-cycle energy consumption. However, this reduction in load must be done in a controlled manner, and cannot be arbitrarily applied when real-time and mission constraints are involved.

We could improve system runtime and make better use of scarce energy if we could limit the energy consumed by less important auxiliary tasks, and instead divert it to the execution of the more critical ones. This is, in a sense, a method of adapting the task set to the available energy resources and providing varying QoS to the tasks to maximize the return on the energy used for their execution. In order to use such a QoS mechanism, we need tasks that are amenable to adaptation and can be executed and different QoS levels with varying energy consumption rates. Application-level energy adaptation for multimedia has been well researched in the literature [21], but as applied to real-time tasks, it is under-explored.

However, in the field of fault-tolerant computing, one commonly deals with real-time tasks that have multiple operating modes. Particularly for large, multiprocessor and distributed real-time systems, methods of executing tasks at reduced service levels requiring less computational resources have been well studied. The idea behind this is that in case of failure, when parts of the system stop working and less resources are available, the system can continue to operate, using alternate task execution modes to ensure that all tasks are still able to run, although at a degraded quality level. This "graceful degradation" is greatly preferred over a system that simply stops working. Of course, any degraded task that requires less computation time also requires less energy to perform those computations. An important component in our EQoS framework, therefore, is to use real-time task adaptation developed for fault-tolerance in the context of energy savings.

To understand how real-time task adaptation is implemented, it is helpful to look at the characteristics of typical real-time (RT) tasks and systems. The canonical model of a RT system includes multiple tasks that are executed periodically with a strict guarantee of their completion by a certain deadline. More specifically, each task i has an associated period, t_i. Every t_i time units, the task is started or released (or invoked). The task also has an associated worst-case execution time (WCET), C_i, and a relative deadline, d_i, measured relative to its release time. As these parameters are necessary for proper real-time scheduling, they are generally explicitly specified for RT tasks. The real-time scheduler guarantees that the task will receive up to C_i units of execution time within d_i time units of each release of this task. The actual execution time for each invocation of a task can vary greatly from the specified WCET, but as long as the tasks use less than their WCETs for each invocation, a provable guarantee of completion within deadlines is provided. Note that in many RT systems, including the classical RT scheduling algorithms [45], it is often assumed that the relative deadline equals the period ($d_i = t_i$, i.e., a task is guaranteed to complete its execution by the time its next invocation is released). Furthermore, this periodic model is used extensively in practice for a wide range of real-time applications, including various embedded control tasks, but is also quite general and has been used to accommodate other types of tasks, including sporadic and aperiodic event handlers [43].

The precise mechanisms employed in executing real-time tasks at degraded service levels depend on the nature of the applications. For control systems, one simple technique employed is to stretch the period of the real-time control task so the frequency of execution is reduced. The control task normally runs at a rate that has been deemed optimal for the system being controlled, providing the desired tradeoff between response time and overshoot limitation, while guaranteeing stability. When processing capacity is reduced due to failures, or we need to restrict computation to conserve dwindling energy resources, the process can be run at a lower periodic frequency, resulting in suboptimal, degraded control. Although performance becomes suboptimal, this may be desirable in order to keep the system operating longer to avoid catastrophes. Of course, this approach has its own limits, as stretching period too much may result in instability, but this is very dependent on the nature of the control system and can be usually determined at the the time the system is constructed.

For other applications, different types of degradation techniques also exist. Imprecise computation [46] models trade off execution time for precision. An example of this is iterative computations — the more iterations performed, the greater the resulting precision. In such tasks, it is possible to reduce the precision and the computational load, and therefore conserve energy resources. Yet other systems may use several completely different algorithms to perform similar tasks. For example, in the case of voice compression, two different CODECs employing different algorithms may produce the same level of compression, but at differing levels of loss or noise, and inversely correlated differences in computing time. In other cases, for a non-critical task, the most degraded service level may simply be not running it at all. Stopping/deleting the unimportant tasks can allow mission-critical tasks to run for a longer period of time when energy is low.

Our intention is not to devise new methods of providing degraded service-levels to real-time tasks, nor enumerate the existing techniques. Rather, we want to take the existing techniques that provide graceful degradation in the case of failures, and apply them to reduce energy consumption within a real-time EQoS framework.

However, we do need to restrict the set of tasks and their service levels to a certain extent. For any set of real-time tasks, one must test whether the particular combination of task requirements is *schedulable*, i.e., all deadlines can be met under a particular scheduling paradigm. In the EQoS framework, each task has multiple service levels, and each service level may have vastly differing real-time requirements, particularly if algorithmic change is involved in the service degradation. Hence, each combination of task service levels needs to be checked and only the schedulable combinations are valid outputs of any adaptation algorithm. This, in the worst case, requires an exponential search through all combinations of task service levels.

To simplify the schedulability problem, we first restrict scheduling to use the *earliest-deadline-first* (EDF) dynamic priority scheduler. This scheduler has the nice property that, assuming negligible scheduling / preemption overheads and independent tasks, all tasks are guaranteed to meet their deadlines as long as the total processor utilization, $\sum C_i/t_i$ over all tasks i, is no greater than one [45]. In the EQoS framework, where each task has multiple service levels, we define u_i^{max} as the largest C/t among all of the service levels of task i. Using this, we state a sufficient condition for schedulability:

$$\sum_{i=1}^{n} u_i^{max} \leq 1, \tag{5.1}$$

where there are n tasks in the system. By restricting task sets to only those that meet this sufficiency condition, we can guarantee that regardless of which service level is selected for each task, the system will always meet the EDF scheduling requirements. With these restrictions, the additional complexity of testing for schedulability is removed, eliminating a potential obstacle to efficient adaptation solutions. On most systems, these restrictions are met by default as schedulability conditions are usually satisfied when all tasks are executing in their highest service level.

5.2.3 Specifying Task Utility

Given a set of real-time tasks that can be executed at degraded QoS levels, it is fairly straightforward to enumerate a valid set of degraded operating modes for each task that will ensure mission-critical goals (e.g., maintaining stability) are met. When determining other characteristics, such as execution times, it is easy to measure energy consumption corresponding to these reduced QoS levels, which can be used in QoS level selection. As real-time tasks are already well-specified with respect to WCET, period, and deadlines, we simply need to specify one additional parameter, the average power dissipation for executing this task, for each QoS level defined for the task. However, to determine the best tradeoff between these task QoS levels, one critical input is needed — a quantification of the value or benefit gained, called *utility*, from running a task at a particular QoS level.

In general, utility is an abstract notion of value or gain, and need not be based on real units. This leaves the actual specification flexible to the type of applications or systems that are designed. Each QoS level for a task is assigned a utility value, reflecting the relative benefits executing different tasks at the various QoS levels defined. Similar notions of a generic utility metric have been successfully used elsewhere [39] for QoS-related optimizations.

With some forms of task QoS degradation, it is relatively simple to assign utility values. With the class of imprecise computation methods that provides "increasing rewards for increasing service" (IRIS) [13], a simple monotonically-increasing function of maximum computation time suffices for utility. Depending on the application, this may be a linear function or a fractional power relationship reflecting decreasing marginal returns.

For real-time control tasks that allow period extension as a graceful degradation method, a mechanism exists for quantifying the effects on the controlled system. Using a control-theoretic measure of *performance index* (PI), one can derive a reward function for various task-periods [78]. In this context, we can generally just use this reward function, scaled by some constant, to find the utility for the various task QoS levels.

For tasks that deal with multimedia, or voice compression, the utility assignment is somewhat trickier. In particular, the output of these applications can only be evaluated in terms of human-perceived quality, which is difficult to quantify. Running a task with reduced computation and energy resources may produce results indistinguishable from the original service level to one user, but seem to incur severe quality loss to a different user. Much research has focused on modeling average human perception of quality in the context of compression, and although such metrics may be used to assign utility, the utility assignment for multimedia tasks remains a difficult and somewhat arbitrary process.

Although in utility assignment there are no particular units of utility or ranges of acceptable values, the assigned utility values need to be consistent among the different tasks. This utility method allows for a wide range of possible task set adaptations when utility is maximized. It is possible to have some tasks run at their maximal (i.e., maximum computational and energy demand) QoS levels, while simultaneously others are at their minimum specified levels. On the other hand, the utility-based specification does have its limits. In particular, in this approach, the utility of running a task is independent of the tasks in the system, so it is not possible to specify constraints such as task A has utility x only if task B is running, or at least two of three tasks A, B, and C must run. However, for most systems, the notion of utility provides sufficient flexibility to define a wide range of preferences or relative benefits of task adaptation.

5.2.4 Maximizing System Utility

The final component of the EQoS framework is the algorithm used to actually select the QoS levels of tasks to be run. In order to obtain the greatest benefits from the limited energy sources, we need to select the QoS levels such that the utility is maximized. Even though all of the task QoS levels, their power requirements, and the utility gained through their executions are specified, there still remains a question of whether it is better to run tasks at minimal levels for a long duration, or execute for a shorter duration at high-utility, high QoS levels, or even a mixture of service levels so the system can run for a specified amount of time. The problem of energy adaptation is somewhat ambiguous, and the optimal solution depends on both the task set and the actual constraints on the system. In the following section, we present in detail the problem of energy-adaptive task QoS-level selection and describe algorithms to maximize utility in an energy-constrained adaptive system.

5.3 Adaptation Goals, Problems, and Algorithms

The goal of EQoS adaptation is to maximize the system utility or value for the given, limited energy resources, subject to mission constraints. By selectively reducing the QoS level provided to individual tasks, the total system load can be reduced, allowing low-level mechanisms to reclaim energy that can be used to run other tasks at high QoS levels, or keep the system functioning for a longer duration. The selection of task QoS levels can be expressed as a constrained utility maximization problem, but the actual algorithms to solve such problems depend on the constraints of the system. In this section, we first formally describe an adaptive system and formulate the selection of adaptation levels as a tractable, utility maximization problem. We then present several algorithms to optimally select QoS levels, as well as some simple heuristics, and then extend these to solve related problems with different mission constraints.

5.3.1 Adaptive System Description

An adaptive task set is formally described as follows. We are given a set of n tasks, $T_1, \ldots, T_n$, of which each task T_i has m_i different QoS levels defined as $T_{i,1}, \ldots, T_{i,m_i}$. For each level $T_{i,j}$, we specify its canonical real-time parameters, period $t_{i,j}$ and WCET $C_{i,j}$, and we also specify a utility value $U_{i,j}$ and an average energy expenditure $E_{i,j}$. Both of these are expressed as per-invocation values. The energy parameter can be measured easily by running the task on the target hardware platform and using standard electronic instruments (e.g., DVM or oscilloscope with current probe).

Given a particular selection of QoS levels, $j_1, \ldots, j_n$, for tasks $T_1, \ldots, T_n$, respectively, the number of iterations executed for task i over a time interval t is in the range, $\left[\left\lfloor \frac{t}{t_i} \right\rfloor, \left\lceil \frac{t}{t_i} \right\rceil\right]$. Assume further that the system dissipates P_{fixed} power when idle, and that the $E_{i,j}$ values indicate the average additional energy consumed for each invocation of task i at service level j, beyond what would have been used for idle. Now, by multiplying the number of iterations by the average energy per invocation for each task and adding these to the fixed energy consumed, we can determine the energy consumed over interval t. Dividing this by t, we obtain a range for the average power, P_{sys} over the interval t:

$$\frac{1}{t}\left(t \cdot P_{fixed} + \sum_{i=1}^{n} \left\lfloor \frac{t}{t_i} \right\rfloor E_{i,j}\right) \leq P_{sys} \leq \frac{1}{t}\left(t \cdot P_{fixed} + \sum_{i=1}^{n} \left\lceil \frac{t}{t_i} \right\rceil E_{i,j}\right)$$

As system runtimes are in the range of minutes to hours, while task periods are on the order of tens of milliseconds, we can safely assume long observation intervals relative to task periods. For large values of t/t_i, P_{sys} converges to:

$$P_{sys} = P_{fixed} + \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}}. \tag{5.2}$$

Similarly, we can determine the rate of utility gain as:

$$\sum_{i=1}^{n} \frac{U_{i,j_i}}{t_{i,j_i}}.$$

Assuming that we have E_{sys} energy units initially available, the total expected system runtime is expressed as E_{sys}/P_{sys}. Multiplying this runtime by the utility gain rate, we can determine the total system utility, U_{sys}, as:

$$U_{sys} = \frac{E_{sys} \sum_{i=1}^{n} \frac{U_{i,j_i}}{t_{i,j_i}}}{P_{fixed} + \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}}}. \tag{5.3}$$

The goal of any EQoS adaptation algorithm is to select the per-task QoS levels, indicated by $j_1, \ldots, j_n$, to maximize U_{sys} subject to system runtime constraints. The actual algorithm depends heavily on these other constraints, so we need to consider each specific type of problem separately in the following sections. In addition, we note that the $E_{i,j}$ values indicate average energy consumption assuming that the processor operates at its maximum frequency and voltage. Employing DVS will reduce the actual power dissipated, and we will explore ways of accounting for its effects in Section 5.3.4.

5.3.2 Known Time-to-Charge Problem

We first look at task adaptation for cases where the system will need to operate on stored energy for a finite time, or the system runtime is bounded by a known value. This is a common scenario, where one knows when primary energy sources will become available to power the system and recharge batteries. An example of this is a solar-powered satellite that has entered the shadow of the planet — given the orbital mechanics, the required time until it emerges out of the shadow can be computed very accurately. Stored energy must be used during this interval, and we would like to adapt the task set to maximize the utility of the system during this interval. We note that in such a system, due to size and weight restrictions, the gradual deterioration of the batteries, software upgrades that change task sets, variations in shadow transition times, and physical inaccessibility, a dynamic system of adaptation is preferable to static techniques or the over-engineering/replacement of the batteries.

In this scenario, we are given the system energy, E_{sys}, and a time, t_{run}. This time can be thought of as the required system runtime or the time until the next recharge for the system. We want to maximize the utility gained during t_{run}, but do not care about execution beyond this time. The reason is that after this time, primary energy sources are available, so there are no longer energy constraints and the system can simply operate using the maximal task QoS levels. Since we are concerned only with utility during this time interval, total system utility simplifies to:

$$U_{sys} = t_{run} \cdot \sum_{i=1}^{n} U_{i,j_i}/t_{i,j_i},$$

assuming that the actual system runtime is at least t_{run}:

$$\begin{aligned} \frac{E_{sys}}{P_{sys}} = \frac{E_{sys}}{P_{fixed} + \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}}} &\geq t_{run} \\ \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}} &\leq \frac{E_{sys}}{t_{run}} - P_{fixed} = P_{budget} \end{aligned}$$

The right side of the inequality can be considered a power budget for task execution that ensures a system runtime of t_{run}. With these derivations, we now need to select per-task QoS levels $j_1, \ldots, j_n$ to maximize the total utility by maximizing the utility rate, $\sum_{i=1}^{n} U_{i,j_i}/t_{i,j_i}$, while ensuring that the system runs for the desired time t_{run} by constraining the total power for task execution, $\sum_{i=1}^{n} E_{i,j_i}/t_{i,j_i} \leq P_{budget}$.

Expressed in this way, the optimization problem reduces to the *multiple-choice knapsack problem* (MCKP) [52], a lesser-known variant of the famous *0-1 knapsack problem.* In MCKP, we have a set of categories, each with a number of non-overlapping items, each of which, in turn, has an associated value and weight/size/cost. Given a knapsack limit, the goal is to select exactly one item from each category to maximize value subject to the knapsack size limit.

Our problem is expressed as an MCKP by treating each task as a category and its set of defined QoS levels as the items within the category. The knapsack size is set to the power budget, $K = E_{sys}/t_{run} - P_{fixed}$. The item values and weights are the utility rates and power dissipation of the tasks, respectively, at each quality level, i.e., $v_{i,j} = U_{i,j}/t_{i,j}$ and $w_{i,j} = E_{i,j}/t_{i,j}$. It now suffices to solve this MCKP to determine the optimal set of task QoS levels, $j_1, \ldots, j_n$, that maximizes the total expected utility of the system with E_{sys} energy units during the time t_{run} until the next recharge.

5.3.3 Solving MCKP

The naive approach to solving this MCKP optimally is a simple state-space search. We systematically iterate through every combination of task QoS level assignment, checking the total power against the power budget, and keeping track of the selection that results in the largest total utility rate. Unfortunately, the search space grows exponentially — if the n tasks have m quality levels each, the search takes $O(m^n)$ time. Therefore, this approach is generally not practical.

There is, however, no known general solution to MCKP that can be performed in polynomial time, since MCKP is an NP-hard problem for the following reason. We can express any 0-1 knapsack problem as an MCKP: for each item a_i in the former, we create an additional item a_i' with weight and value equal to zero. Then, each category consists of the item and its zero value counterpart, i.e., $\{a_i, a_i'\}$. The knapsack size is the same for both problems. If a_i is included in the MCKP solution, it is also in the 0-1 knapsack solution; if a_i' is included, then a_i is not in the 0-1 knapsack solution. With this mapping, any polynomial solution to MCKP can be used to get a polynomial-time solution to the NP-hard 0-1 knapsack problem. Hence, MCKP is NP-hard as well.

However, if we assume that the weights (i.e., power) can be expressed as integers, then we can obtain a pseudo-polynomial-time optimal solution using dynamic programming (DP) techniques. We first solve trivially the MCKP containing just one category (task) for all possible knapsack sizes (power budgets), which are also integers. Using the optimal solutions to this subproblem, we can find the solution to MCKP for the first two tasks in linear time for each possible power budget. Repeating this process of building on the previous partial solutions, we obtain the solution to MCKP with all n tasks, for all power budgets. This has a pseudo-polynomial-time complexity of $O(nmk)$, assuming there are n tasks with m QoS levels each, and a maximum power budget (knapsack size) of k. Unfortunately, as k can be large, the time required may be significant. Furthermore, DP requires significant amounts of memory as well — the space complexity is $O(nk)$. This latter may make it impractical for small embedded controllers, which are typically memory- as well as processing- and energy-constrained.

One final approach to the optimal solution is to use a branch-and-bound (BB) algorithm. This involves a depth-first traversal of the decision tree, but only promising branches are visited, greatly reducing the time relative to simple exponential search. Each level of the decision tree is associated with a task, and each node at level i has m_i branches, corresponding to the m_i QoS levels defined for T_i. A fast bounding algorithm is used to compute an upper bound to the total value that is possible if a particular branch is taken. Only branches with an upper bound on value greater than any seen so far are visited.

This technique requires a fast algorithm to compute a good upper bound on the value of MCKP. We use the linear relaxation of MCKP called linear MCKP (LMCKP) [67, 92], which can be solved quickly. LMCKP allows making fractional selections, which permits interpolation between two defined QoS levels of a task, e.g., 0.75 $T_{2,3}$ and 0.25 $T_{2,4}$, providing weighted average values for the power and utility rates. To solve LMCKP, we start with all tasks at minimal service levels. We enumerate all of the possible "upgrades," or changes in selection from a lower QoS level to a higher one, for each task, and then sort these for all tasks by the utility-change to power-change ratios. We apply the upgrades in order, the one with the highest ratio first. If applying an upgrade exceeds the power budget, we apply it proportionally to the available power, resulting in a fractional selection for one task, a fully-used budget, and task selections providing the greatest marginal utility. This LMCKP solution is optimal, and is guaranteed to always give a value no lower than the optimal discrete MCKP solution, so it can be used as a fast upper bound on MCKP.

The BB algorithm produces an optimal solution and is not affected by large knapsack size and is not limited to integers as with DP. The drawback is that there is no guarantee of effective pruning of the search space, so this may, in the worst case, require very long execution time as with an exponential search.

To overcome this drawback, we also consider a couple of simple heuristics that have very short execution times. The first heuristic uses the solution to LMCKP, which was used as the upper bound in the BB algorithm. We simply drop any fractional selection, replacing it with the lower of the fractionally-selected QoS levels, to produce a valid, though not optimal, discrete solution.

A slightly better solution is achieved through a greedy algorithm. This starts out just like the linear approach, "upgrading" selections in order of the largest utility-change to power-change ratios. Rather than using a fractional selection to fill out the knapsack, the greedy heuristic continues to look through the sorted list, performing any possible QoS-level upgrade. This should result in a total utility no lower than the linear heuristic.

The execution times for these are very low, both incurring a linear complexity, in addition to any overhead of sorting the QoS-level upgrades. This sorting is needed only when

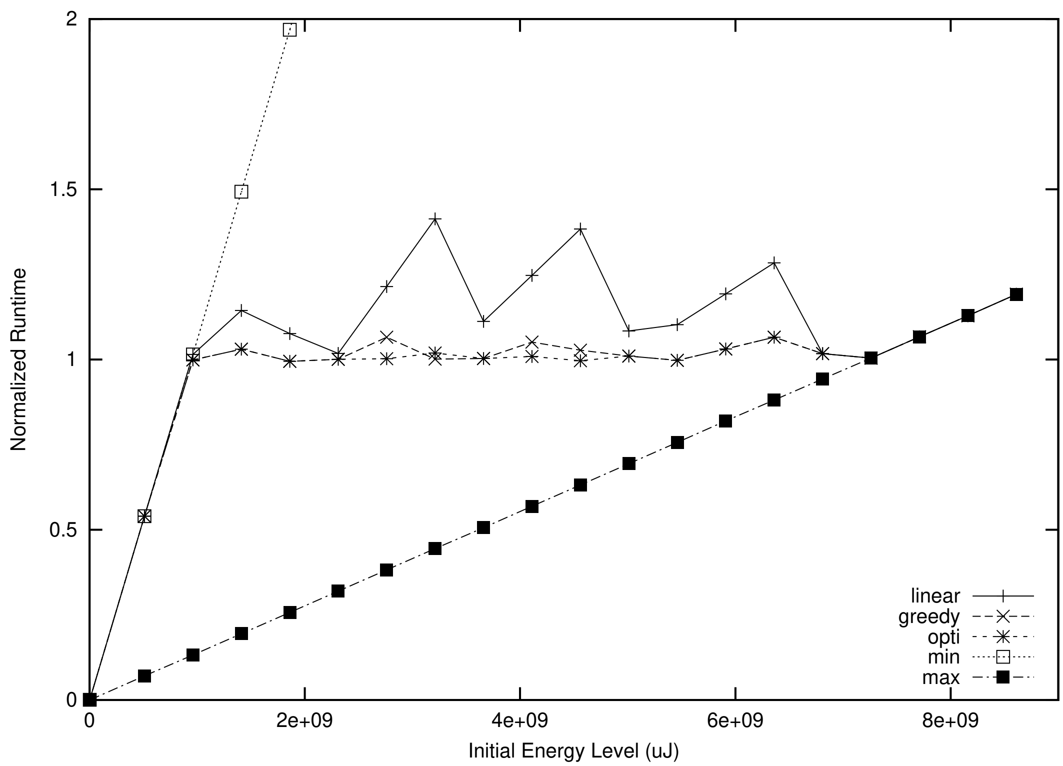


Figure 5.2: Example showing effects of adaptation on runtime, normalized to t_{run}.

the actual task set changes, so this overhead is not always incurred. Unfortunately, there is no guarantee on how close to optimal the resulting solutions are with these heuristics. It is possible to construct task sets, for which these heuristics produce arbitrarily poor solutions relative to the optimal algorithms. However, for most realistic task sets, the heuristics will produce reasonable solutions, albeit with some deviation from optimality.

To illustrate the effects of these different adaptation algorithms, we show in Figure 5.2 the resulting system runtime normalized to the desired runtime for a sample task set while varying initial energy, where “opti” refers to the optimal solution obtained from the DP and BB approaches, and “linear” and “greedy” refer to the heuristic solutions. For comparison, we also include a non-adaptive use of the minimal and maximal QoS levels for the tasks, labeled as “min” and “max”, respectively. Adaptation produces selections between these extremes and maintains system runtime near the desired value of t_{run}. At the extremes of the initial energy range, however, we cannot adapt any further, and therefore overlap the minimal or maximal curve as initial energy is varied. The total utility until the known time-to-charge, t_{run}, for this example task set is plotted in Figure 5.3. We present the specifics

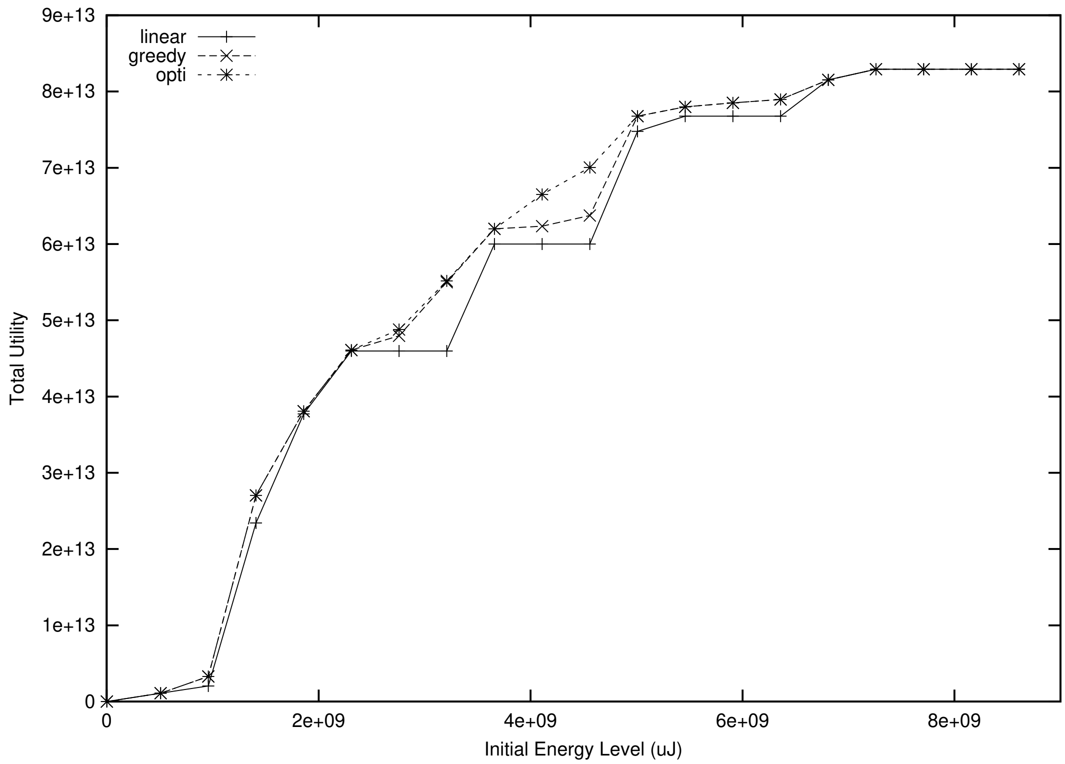


Figure 5.3: Example task set total utility gain comparison using adaptation.

of our simulations and detailed evaluations of these algorithms in Section 5.5.

5.3.4 Effects of DVS

Thus far, we have not considered the effects of DVS on adaptation. In particular, DVS techniques allow for greatly reduced per-cycle energy costs when the processor is lightly-loaded. This translates into a greatly-increased runtime for the system. Figure 5.4 shows the system runtime normalized to desired runtime after adaptation of the example task set used in the previous figures, but in addition, an aggressive real-time DVS algorithm is employed. The system in this example has 3 voltage-frequency combinations shown in Table 5.2. Compared to the previous results, we see much longer system runtimes, particularly when the system has very low utilization.

Of course, the problem we are dealing with — maximizing utility until a known recharge time — does not directly benefit from the extended runtime. Instead, We would rather use the benefits of DVS to run tasks at higher-power, greater-utility QoS levels. Unfortunately,

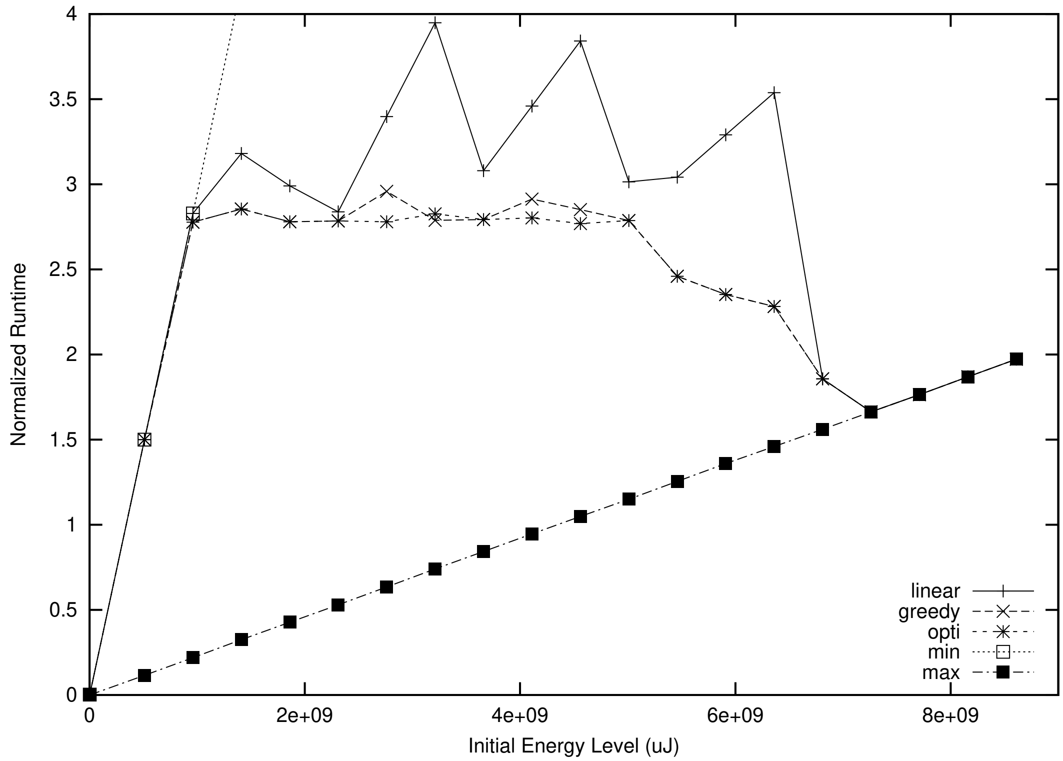


Figure 5.4: Example showing effects of DVS on runtime.

the exact effects of DVS depend on the actual execution times for each invocation of all tasks, which is a random component and cannot be predicted *a priori*. As a result, we cannot take DVS directly into account in the adaptation algorithms. Instead, we would like to compensate for the effects of DVS and make the system run to the desired time with higher utility by providing higher QoS to the tasks.

To do this, we introduce the concept of the *idealized DVS response*. Based on our prior work in DVS algorithms [65], we note that advanced DVS algorithms that aggressively reclaim unused slack time achieve energy performance close to an easily-computed lower bound. Figure 5.5 shows the relationship between average power and processor-capacity utilization for idealized DVS for the particular settings in Table 5.2. The idealized response reflects energy-per-cycle proportional to the voltage squared when the utilization equals the normalized frequency settings that are available. For utilization values between these, the average energy-per-cycle is a linear interpolation of these. Idle cycles are assumed to consume no energy (i.e., a perfect processor halt mechanisms), so the solid line in Figure 5.5 is obtained by multiplying the average (normalized) energy-per-cycle by the processor uti-

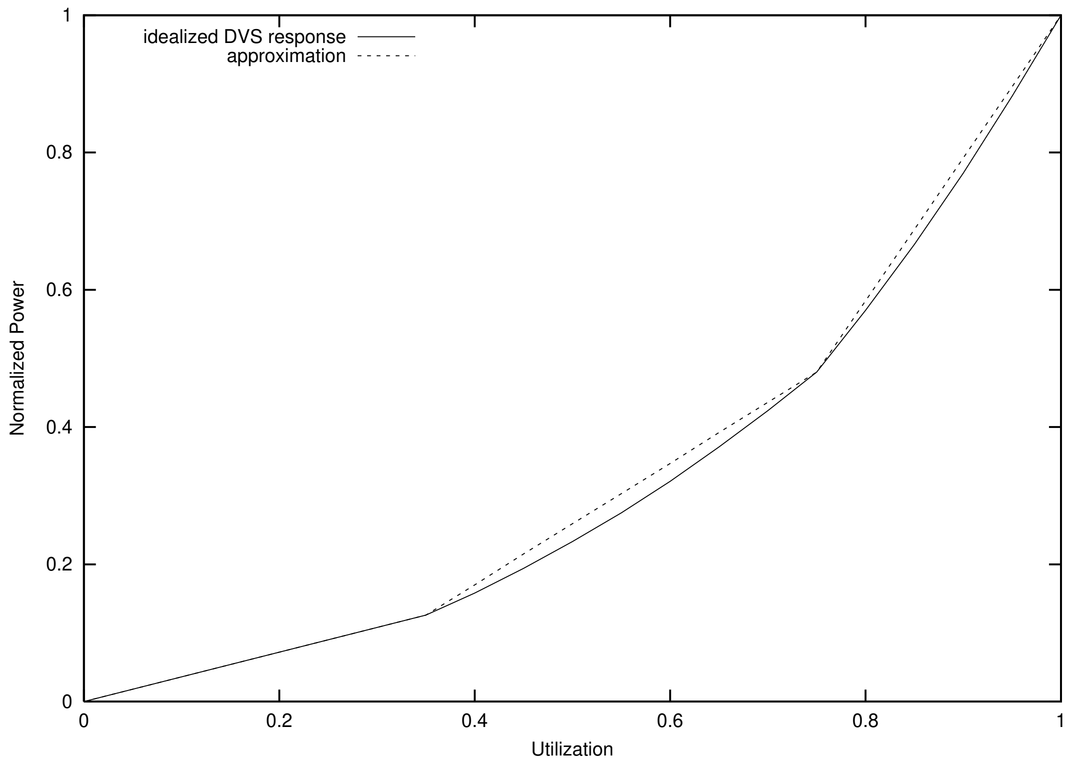


Figure 5.5: Relationship between utilization and power for ideal DVS.

Normalized frequency	Normalized voltage
1.0	1.0
0.75	0.8
0.35	0.6

Table 5.2: Normalized frequency and voltage settings for DVS.

lization, which equals the normalized number of non-idle cycles per second.

We can use the inverse relationship to determine the processor utilization that results in a particular normalized power budget. Call this inverse relationship the *compensation function*, or $F_{comp}()$. As the previously-discussed adaptation algorithms do not use utilization directly, we can convert the utilization value to a power value by multiplying it by $(P_{max} - P_{fixed})$, the maximum additional power dissipation at maximum system utilization, thus obtaining the power dissipation expected for the target processor utilization if DVS were not employed:

$$P_{comp} = (P_{max} - P_{fixed})F_{comp}\left(\frac{P_{budget}}{P_{max} - P_{fixed}}\right).$$

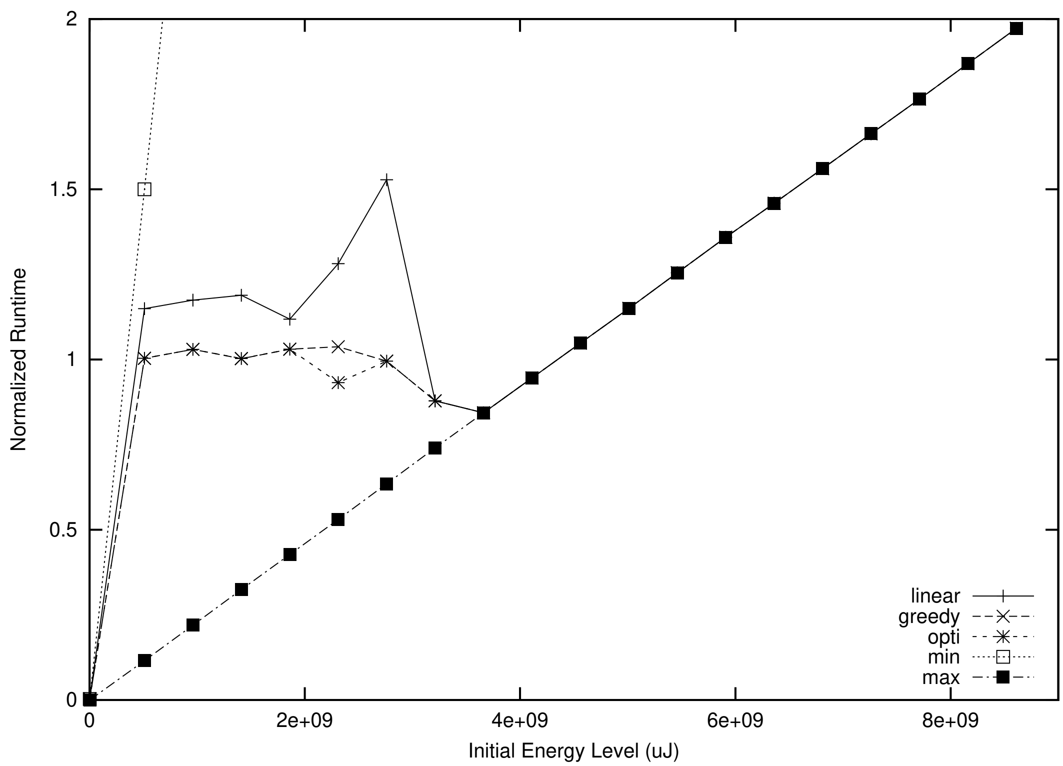


Figure 5.6: Example showing effects of DVS on runtime, normalized to t_{run}, when compensation is used.

We now feed P_{comp} as the power budget parameter to the adaptation algorithms. In other words, we use F_{comp} to adjust the power budget to compensate for the effects of DVS. In fact, we simplify this a little bit by using a piecewise-linear approximation of F_{comp}, the inverse of the dotted line in Figure 5.5. By accounting for DVS in this way, we implicitly make the assumption that all power dissipated for executing a task is scaled when DVS is employed. This, of course, is not entirely accurate, since the energy for task execution, the $E_{i,j}$ terms, include consumption in buses, main memory, and I/O devices, in addition to the voltage-scaled CPU. However, since processor power generally dominates non-idle time energy expenditure, this approximation is generally acceptable, especially since F_{comp} is based on an approximation to an idealized response anyway. Figure 5.6 shows the same task set as above under adaptation and DVS, but with a compensated power budget. The system runtimes are now close to the desired runtime.

In practice, DVS mechanisms cannot actually provide the idealized response, as actual task execution times are random and cannot be expected to hold perfectly to their averages over short runs. Furthermore, some distributions and combinations of tasks simply do not work as well with DVS as others (e.g., due to on-off distribution of computation time). Due to this, combined with the fact we approximate all task energy as scaled by DVS, it is very likely that this method of adjusting the power budget for adaptation may over-compensate for the true effects of DVS. This is apparent in Figure 5.6, where the normalized runtimes dip slightly below 1.0 for the adaptation curves just before they merge with the maximum service curve. It is therefore best to use this compensation mechanism as a first approximation, then adapt the task set again at later times. We note that if the adaptation algorithm uses the DP approach, no additional work is really needed for subsequent adaptations, since DP computes optimal solutions for all possible power budgets with the given task set.

5.3.5 Applicability to Other Adaptation Problems

So far, we have considered the optimization of utility when we know the time until the primary power source becomes available. In addition to this scenario, we can also look at several other types of optimization problems.

First, one can consider maximizing system runtime. This is a very trivial problem — one simply needs to run each task at its minimal QoS level to maximize the runtime. As this is not a very interesting problem, we will not consider it any further.

A more interesting problem is the unconstrained maximization of utility, i.e., regardless of runtime. In this case, we need to deal with the unconstrained complex formulation of system utility presented in Equation 5.3. Despite this complexity, this adaptation problem is still tractable. In particular, we can solve this using the solution to the known time-to-charge problem discussed earlier. We use the adaptation algorithms to find optimal solutions for all possible power budgets, and then compute the resulting runtime and total utility for each power budget. We now simply select the power budget that gives the maximum expected utility. We note that with the DP approach, there is little additional work here, since the DP algorithm already finds optimal solutions for all possible power budgets (knapsack sizes) anyway. We simply have to determine the runtime and total utility for these. In addition, we can obtain a suboptimal approximation, possibly at a lower computational overhead, by evaluating the greedy heuristic for all possible power budgets.

A third variant is the maximization of utility subject to a required minimum runtime.

Unlike the known time-to-charge problem, additional runtime beyond the requirement does contribute to the total benefits gained. The simplest, most direct solution to this problem is to implement the algorithm to solve the unconstrained utility maximization problem, but limit the final selection to the power budgets that provide at least the required system runtime.

5.3.6 Dealing with Dynamic Systems

Thus far, we have basically treated the task set as static. In general, real-time systems, especially in embedded systems, tend to have static or very infrequently changing task sets, so dealing with dynamic tasks is not a core concern of our adaptation. With the assumption of infrequent changes to the task set, the simplest method of dealing with dynamic tasks is to simply redo the energy adaptation on each task set change. As there are usually additional overheads associated with adding or deleting tasks (particularly when admission control is employed), this simply adds one additional computation during these changes.

With the branch-and-bound algorithm, we need to redo the entire search, incurring the full computational overhead on each task set change. In case of the DP approach, adding a task is relatively simple, since the existing solutions for n tasks are simply reused as the partial solutions for $n+1$ tasks, and we incur only an $O(mk)$ overhead, where the new task has m QoS levels defined, and the maximum power budget is k. However, when deleting a task, all partial solutions may be invalidated, so the complete computation with $O(nmk)$ overhead may be needed. If the heuristics discussed earlier are used, the runtime overhead is too small to worry about, as long as an efficient sorted queue structure is used to keep the sorted QoS "upgrade" lists.

We note that in addition to performing adaptation when the task set changes, it is best to perform these adaptations periodically. This will ensure that the errors introduced by inaccurate or imprecise task energy values, or the overcompensation for DVS effects are fixed and we can achieve close to the desired runtime with maximal utility.

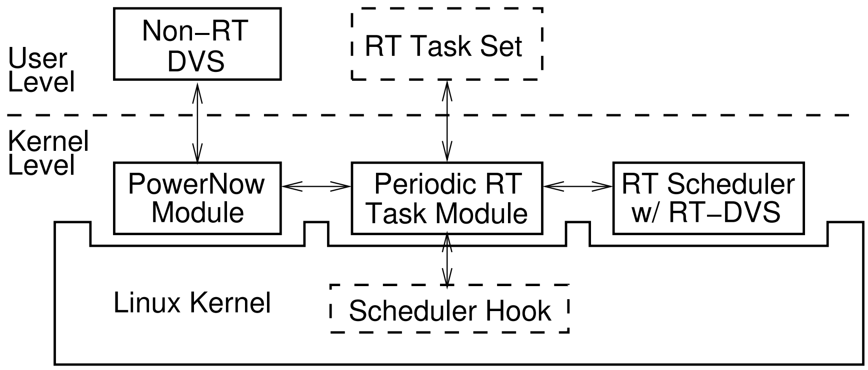


Figure 5.7: Software architecture of EQoS implementation for Linux.

5.4 Implementation

We have developed a working implementation of the proposed EQoS framework. It is built on top of an existing RT-DVS system [65], which provides a periodic real-time task model and support for DVS on top of the Linux operating system. DVS requires hardware support, and our implementation currently works on notebook computers with AMD Mobile Athlon, Duron, and K6-2 processors that implement AMD's PowerNow! voltage and frequency scaling support. Due to the changes introduced in Linux 2.4 kernels, the current implementation only works on the 2.2 kernels.

Figure 5.7 depicts the overall software architecture of the EQoS implementation. The proposed EQoS is implemented as modules that extend the Linux kernel. The core of the system is a module that provides the periodic real-time task model on top of Linux, interacts with tasks, and provides connection points to "plug in" various extensions. Various scheduler modules can plug in to this module, and implement one of several real-time scheduling policies, both with and without DVS support. The actual hardware control of frequency and voltage is done through a third module, the `PowerNow` module, which abstracts the specifics of the hardware.

Our EQoS adaptation algorithms extend this core RT-DVS through the `Adaptation` module. Although these adaptation mechanisms are ideally implemented as middleware, it was simplest for us to create another kernel module to perform this functionality. A fifth module, `batmon`, is intended to measure the available energy in the battery. It can be interfaced with energy monitoring mechanisms such as SmartBattery API [74]. For our implementation, we instead use it as a user interface to specify initial energy values. It also allows us to simulate controlled partial system power failures to see how the system adapts.

All task interactions are performed through the Linux `/proc` file system. A new RT task will open a special file and write its parameters. These include the number of valid QoS levels, and for each level, the real-time parameters (i.e., period and WCET), average energy, and utility. Once registered, a real-time task will be blocked by our module and invoked periodically, according to the specifications. A task indicates it has completed the current invocation by writing to the special file, and the task is immediately blocked until its next execution period. When the task is re-invoked, the return value from this write operation will indicate the QoS level selected so the appropriate degradation mechanisms can be employed.

The adaptation algorithms select QoS levels for all of the tasks in the system. These algorithms are executed whenever a new task is added to the system, or when a task is removed. In addition, they are also run when the `batmon` module indicates a new measure of energy capacity. In our current implementation, we use a dummy battery capacity measure, so the algorithms are run when the user supplies a new power budget through the `/proc` interface to the `batmon` module.

Our prototype EQoS implementation for Linux 2.2.x kernels is publicly available for those interested [64].

5.5 Evaluation

In order to evaluate the benefits of EQoS and the relative performance of the adaptation algorithms for a wide range of task sets, we have developed a parametric simulator that models various aspects of energy-adaptive real-time systems. This allows us to explore a large space of multidimensional task set properties very quickly and determine the expected range of behavior for our adaptation mechanisms. In addition, we also perform actual measurements on our implemented system, and validate some simulation results on a more limited set of tasks.

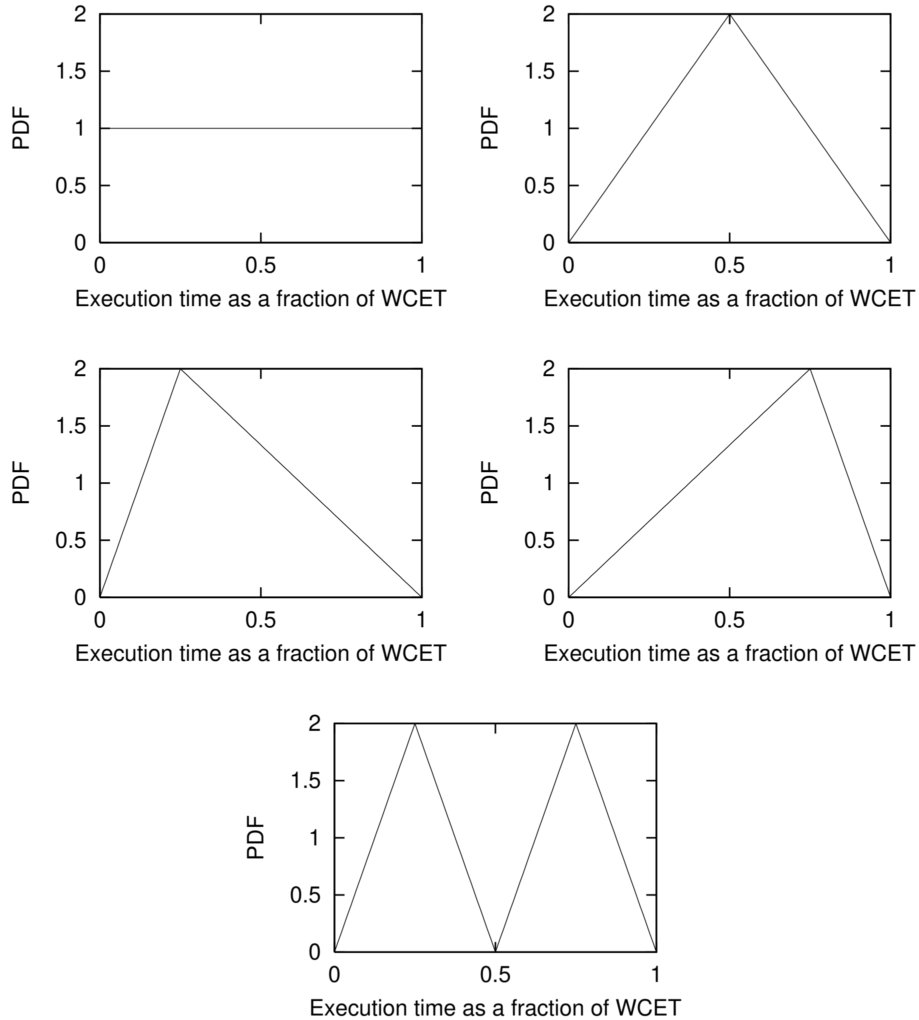


Figure 5.8: Probability distributions for actual execution time of tasks expressed as a fraction of WCET used in the simulator.

5.5.1 Simulation Methodology

Our simulator can model the energy consumption of a processor with voltage and frequency scaling hardware. This simulator can use a variety of real-time scheduling policies, as well as several different real-time DVS mechanisms. The simulator takes an input of various parameters describing the simulated hardware, scheduling policies, and a task set with multiple QoS levels defined. In addition, the total available energy as well as the desired system runtime is supplied. The simulator applies one of our EQoS adaptation algorithms, and then simulates the execution of the task set on the modeled processor to determine the total system runtime and the utility gained. The simulation is repeated for each of our other adaptation algorithms.

The simulator assumes that the energy-per-cycle of computation is constant for a given operating voltage. This value is chosen to be the average energy-per-cycle and is scaled by a normalized voltage squared term to account for the reduced power dissipation from reduced voltage operation when DVS is employed. This simplifies the simulation, since the type of instruction executed is not taken into account, so instruction traces are not needed. Instead, it suffices to count the number of execution cycles between scheduling events to determine energy consumed, allowing for an event-driven simulation rather than a much slower cycle-by-cycle trace simulator. Furthermore, we assume that idle cycles dissipate negligible energy, i.e., the system employs an efficient `halt` instruction instead of idle loops. Only the processor energy dissipation is considered here. When DVS is used, we assume the normalized voltage and frequency combinations described earlier in Table 5.2. Finally, since we are not interested in comparing real-time schedulers or DVS mechanisms, we restrict the system to earliest-deadline-first (EDF) scheduling and the *laEDF* RT-DVS mechanism [65], although one can also trivially apply other scheduling and DVS policies.

We use random real-time task sets to simulate a wide variety of tasks and evaluate adaptation across a wide range of initial energy states. When creating these random task sets, we first consider only the characteristics at maximum quality of service. To reflect the wide range of task periods found in real-time systems, each task has an equal probability of having a short (1–10 ms), medium (10–100 ms), or long (100–1000 ms) period. Within each range, the task periods are selected according to a uniform distribution. WCET for the tasks are assigned according to a similar three-range uniform distribution, and then scaled by a constant to maximize worst-case processor utilization while ensuring the task set is EDF-schedulable (i.e., $\sum C_i/t_i \leq 1$) [45]. Each task is also assigned one of five execution time distributions for its actual execution times to be used in the simulation. Figure 5.8 shows the five possible probability distributions of task execution time, expressed as a fraction of WCET. Assigned is a random utility value, as well as an average power dissipation value, computed from the average execution time and task period.

Next, we generate the task characteristics for degraded service quality. For each task we define a random number of QoS levels, up to a user-specified maximum. We model three different mechanisms of degraded execution: period extension, imprecise computation,

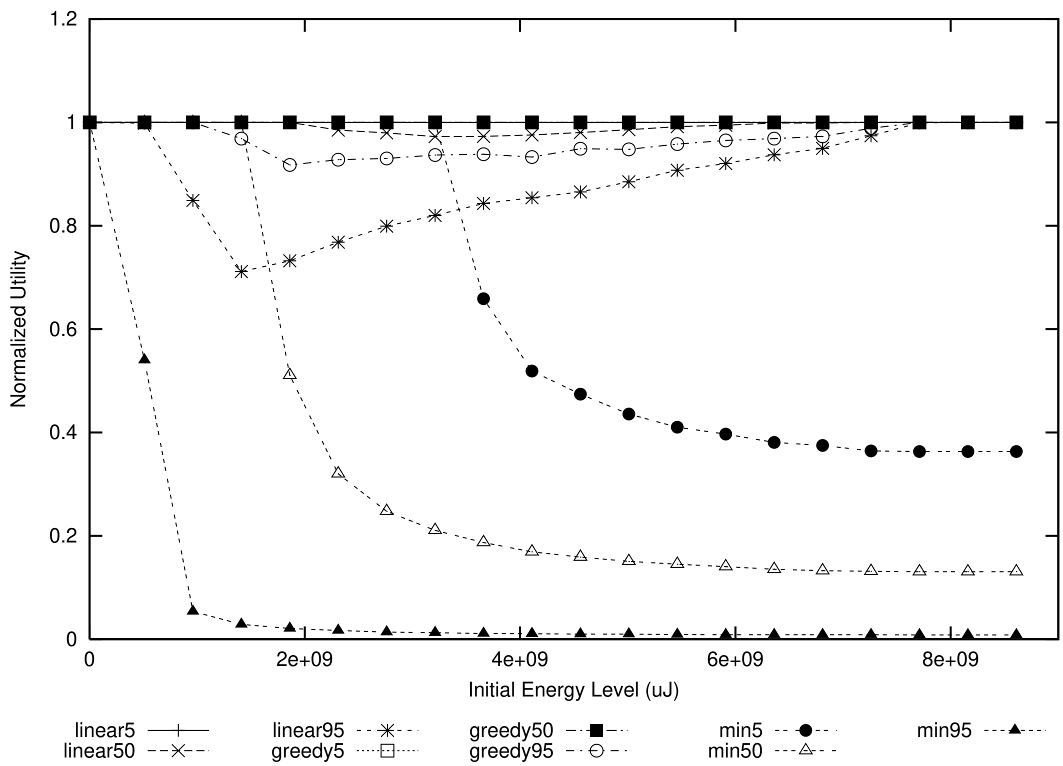


Figure 5.9: Utility with adaptation, normalized to optimal.

and algorithmic change. To model period extension, WCET and execution-time distribution are kept the same, while the period and average power are elongated and scaled down, respectively, by a uniform random variable within the range (1.0, 2.0), and the utility scaled down by a random number for each degradation of QoS levels. For imprecise computation, WCET, average power, and utility are similarly changed. An algorithmic change reflects a change in all of the terms including execution-time distribution, so all are randomly selected such that the worst-case utilization (C_i/t_i), average power, and utility decrease for each additional QoS level defined. Finally, approximately half of the tasks are assigned an additional QoS level, in which they incur zero power and produce zero utility, reflecting that the tasks are non-critical and may be stopped/dropped altogether if the energy budget warrants it.

We create 1000 task sets, each with 10 tasks, and each of which, in turn, has up to 5 QoS levels. We run each of the adaptation algorithms on all of these task sets to generate the following results. The simulations use a desired runtime (t_{run}) of 10 minutes, and vary the initial energy, which is specified in μJ. The processor is assumed to dissipate a maximum

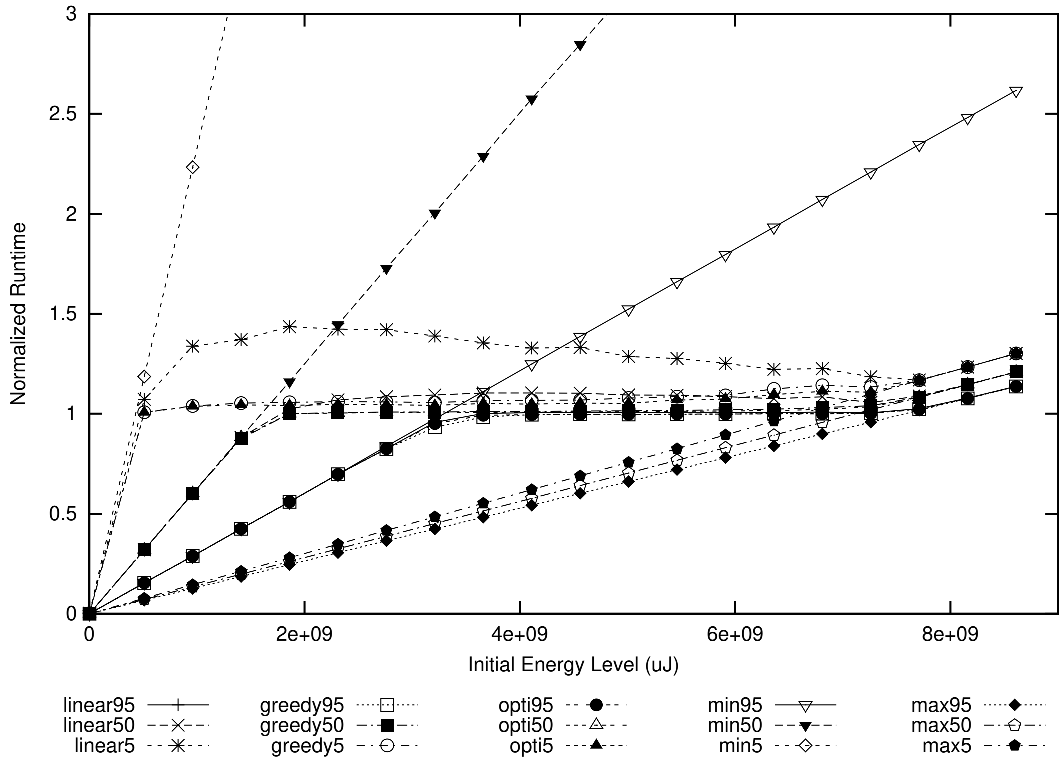


Figure 5.10: System runtime with adaptation, normalized to t_{run}.

of 25 W, which is comparable to most current laptop processors. Note that the adaptation is performed only once at the beginning of each run.

5.5.2 Simulation Results

We have performed extensive simulations to evaluate the benefits and relative performance of EQoS adaptation algorithms. We first compare the algorithms to see how well they adapt task sets to maximize utility, and compare their overheads. We then evaluate the effects of DVS and our accounting mechanism.

Comparison of Adaptation Algorithms

Figure 5.9 shows the relative total system utility provided under various adaptation algorithms. In this figure, we normalize the results for the algorithms relative to the optimal solutions (produced by the DP or BB algorithm). To show the range of results, we plot the 5^{th}, 50^{th} (i.e., median), and 95^{th} percentile normalized utility values from the 1000 different

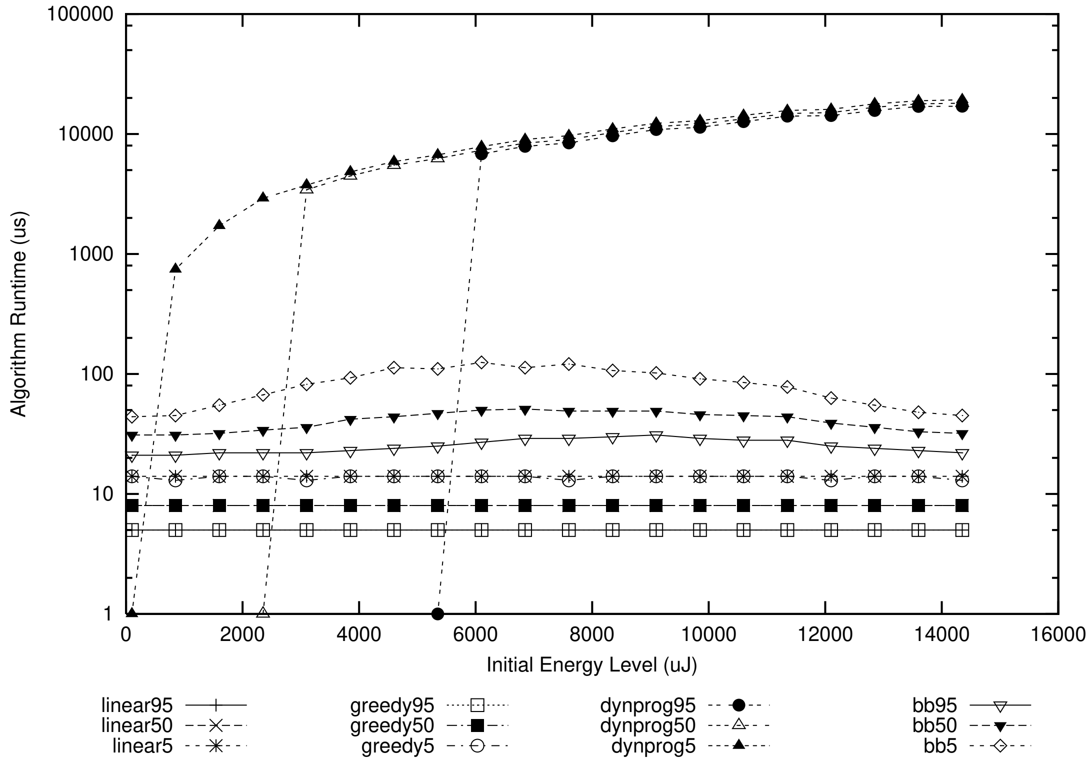


Figure 5.11: Execution overheads of adaptation algorithms.

task sets run for each of the initial energy values. Also for comparison, we show the utility of simply running all tasks at the minimum QoS levels. The greedy heuristic, in particular, performs very close to the optimal solutions, being within 0.9 of the optimal for at least 95% of the task sets for all initial energy values.

The suboptimal adaptations result in runtimes longer than the desired t_{run}, but recall that for the known time-to-recharge scenario, only the utility until time t_{run} is of value. The actual runtimes achieved for this set of experiments is shown, normalized to t_{run} (in this case 600 seconds), in Figure 5.10. Again, we use percentile plots to indicate the range of results. Plots for running tasks at the minimal and maximal QoS levels are shown for comparison. Here, the optimal and greedy methods always result in close to the desired runtime, but the linear heuristic may vary quite considerably.

Figure 5.11 shows the execution time overheads for the different adaptation methods measured while running our experiments. Note that this is plotted on a log scale. All of these were measured on an AMD Athlon XP1500 machine. DP has very consistent, very long execution times, on the order of a few milliseconds. This is due to the large range

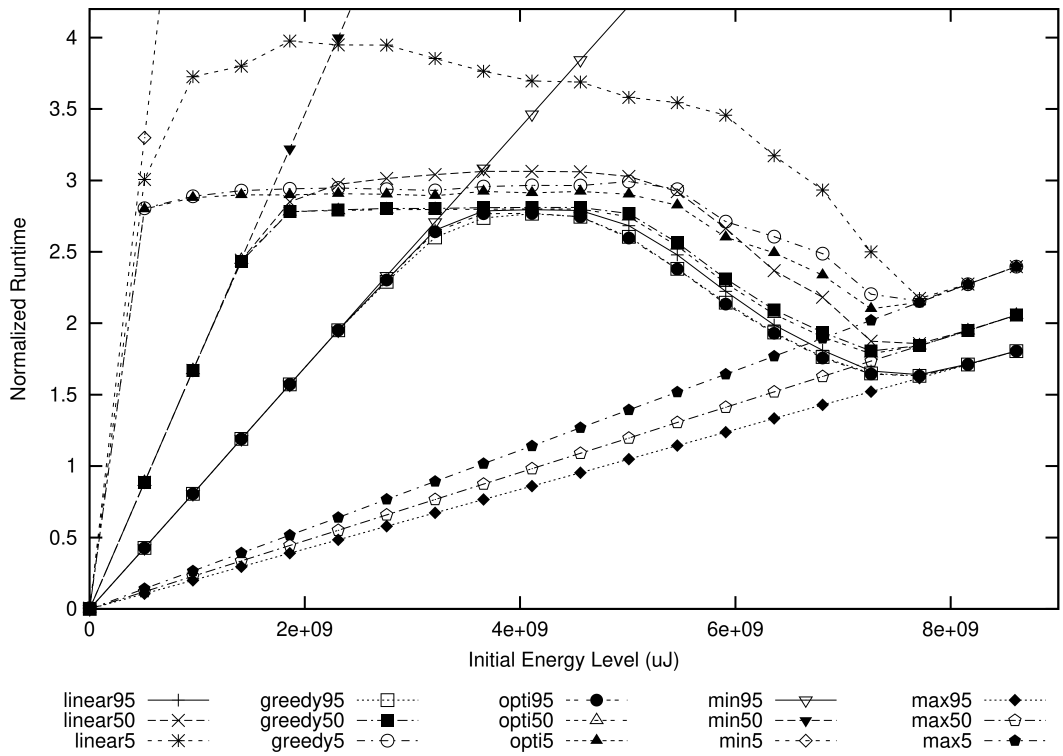


Figure 5.12: System runtime with adaptation, RT-DVS, normalized to t_{run}.

in power values in our task sets, resulting in a large table of partial solutions. The two heuristics, greedy and linear, are very fast, typically requiring under 10 μs. Most of this time is spent sorting the possible task QoS upgrades, so the actual selection heuristics take only about 2 μs. For these task sets, with 10 tasks and up to 5 QoS levels per task, BB incurs fairly low overheads, a few hundred μs. However, its execution overhead varies greatly, and there is no guaranteed time bound, so in the worst case it may degenerate to an exponential search that can take on the order of tens of minutes with these parameters.

Compensating for the Effects of DVS

We repeat the experiments with the same 1000 task sets, but now also employ an RT-DVS scheduler using the voltage and frequency settings in Table 5.2. The energy-conserving DVS greatly reduces the average per-cycle-energy consumption of the adapted task sets. The results are plotted in Figure 5.12. Again, we plot the 5^{th}, 50^{th}, and 95^{th} percentile system runtimes that are normalized with respect to the target runtime among

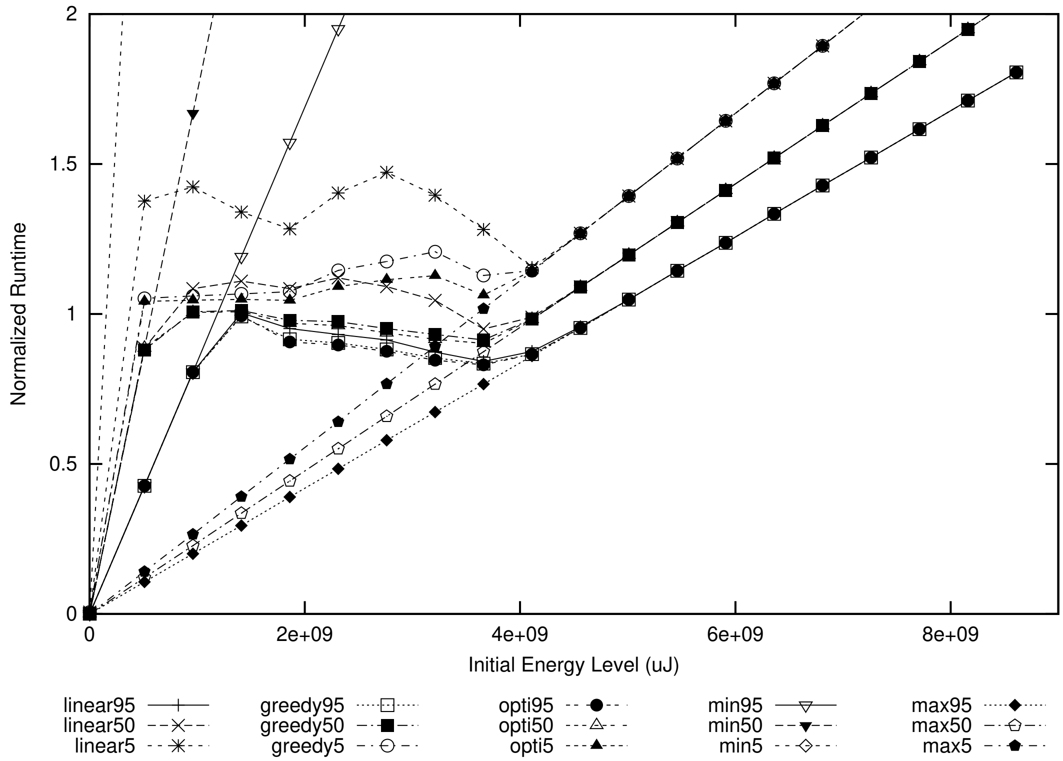


Figure 5.13: System runtime resulting from adaptation with compensation for RT-DVS, normalized to t_{run}.

the 1000 task sets for each initial energy value. In general, we see a significant increase in total runtime, often close to 3 times the desired t_{run}. However, since only the utility of task execution until t_{run} is counted, this does not directly benefit the system.

We repeat the experiment again, and this time employing the compensation mechanism discussed in Section 5.3.4. By selecting higher-energy tasks, we now reduce the runtime to be closer to t_{run} and increase the utility gained. Figure 5.13 shows the resulting runtimes. For the most part, the resulting runtimes for our adaptation algorithms straddle close to 1.0, indicating that a runtime close to t_{run} is achieved. Figure 5.14 shows the change in utility due to compensation. The utility with compensation mechanism enabled is shown normalized to the utility when compensation is disabled. Although the utility gain will depend heavily on the utilities assigned to tasks at degraded QoS levels, there is a very large and consistent change with our randomized task sets. It is interesting to see that we can achieve largest benefits when the energy in the system is relatively low, which is exactly when the energy per computation is most precious. Furthermore, the distribution of utility

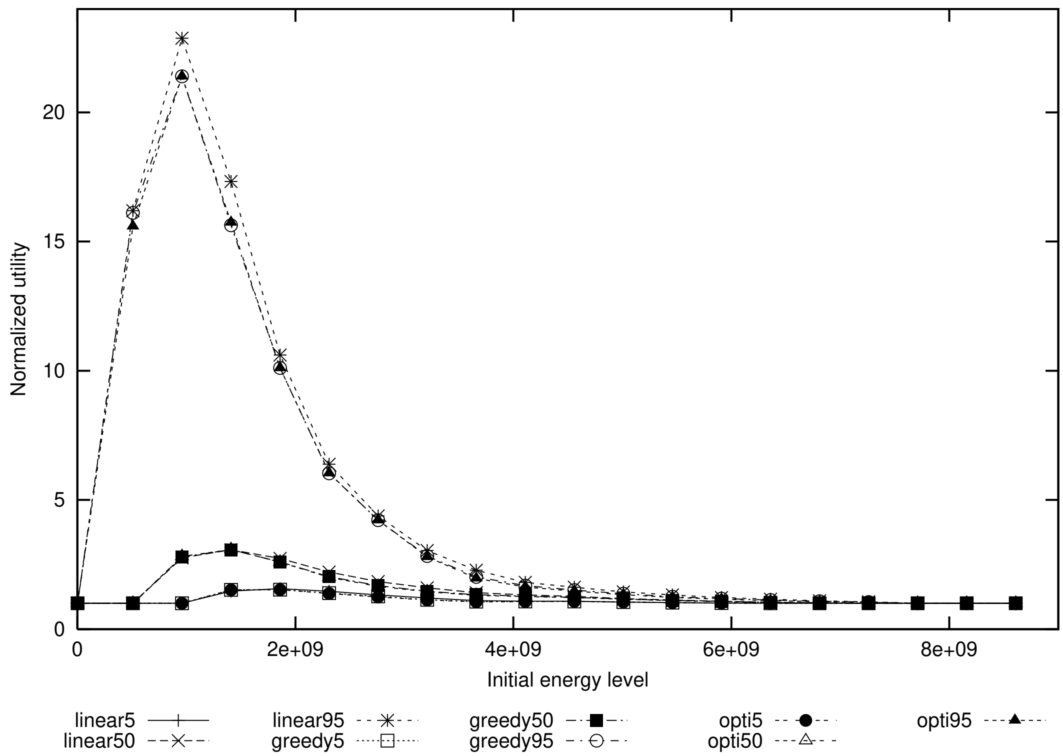


Figure 5.14: Utility with RT-DVS compensation normalized with respect to utility value without RT-DVS compensation.

gain is fairly independent of the adaptation mechanism used, as indicated by the nearly overlapping percentile curves.

One important observation about DVS compensation from Figure 5.13 is that, although on average it achieves very close to the desired runtime, there is a high probability that it will over-compensate and have runtime less than t_{run}. It is, therefore, particularly important that, when DVS and compensation are used, the adaptation is not simply computed once. Rather, the system should be re-adapted periodically to ensure that over-compensation is corrected and the desired runtime, t_{run}, is achieved.

5.5.3 Experimental Measurements

In addition to the extensive simulations, we also evaluate the EQoS framework through measurements of power dissipation using our working implementation on top of Linux OS. The platform for our experiments is a Compaq Presario 1200Z laptop (AMD Mobile

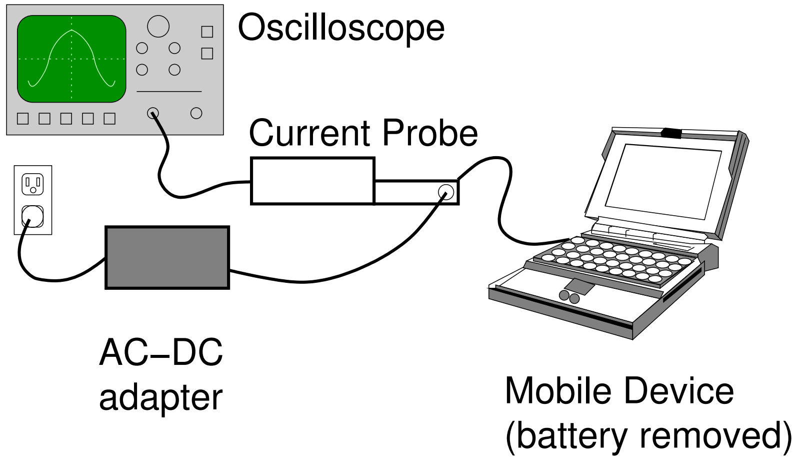


Figure 5.15: Power measurement on laptop implementation.

MHz	$\leq$ 500	600	700	800	1000
Volts	1.20	1.25	1.30	1.35	1.40

Table 5.3: DVS settings for 1GHz Mobile Athlon [3].

Athlon, 1 GHz). By removing the battery and connecting the AC adapter through a current probe attached to a digital oscilloscope, as shown in Figure 5.15, we are able to accurately measure power dissipation of the system. The machine draws 16–18 W while idle (screen and disk on, no active processes), and peaks at approximately 41 W with full processor load.

This processor incorporates DVS support in the form of AMD's PowerNow! [3] technology. The recommended voltage and frequency settings are shown in Table 5.3. Note that, based on this and the quadratic relationship between voltage and energy, at most 27% reduction in per-cycle energy cost can be expected.

QoS level	Period (ms)	WCET (ms)	Avg. CPU Power (W)	Utility
0	22.0	0	0	0
1	22.0	1.45	0.77	100
2	22.0	2.5	1.78	150
3	22.0	3.7	2.72	190
4	22.0	4.3	3.35	220

Table 5.4: `rt-lame` task characteristics at various QoS levels. Note that WCET and power are specified for 1.0 GHz, 1.4 V operation.

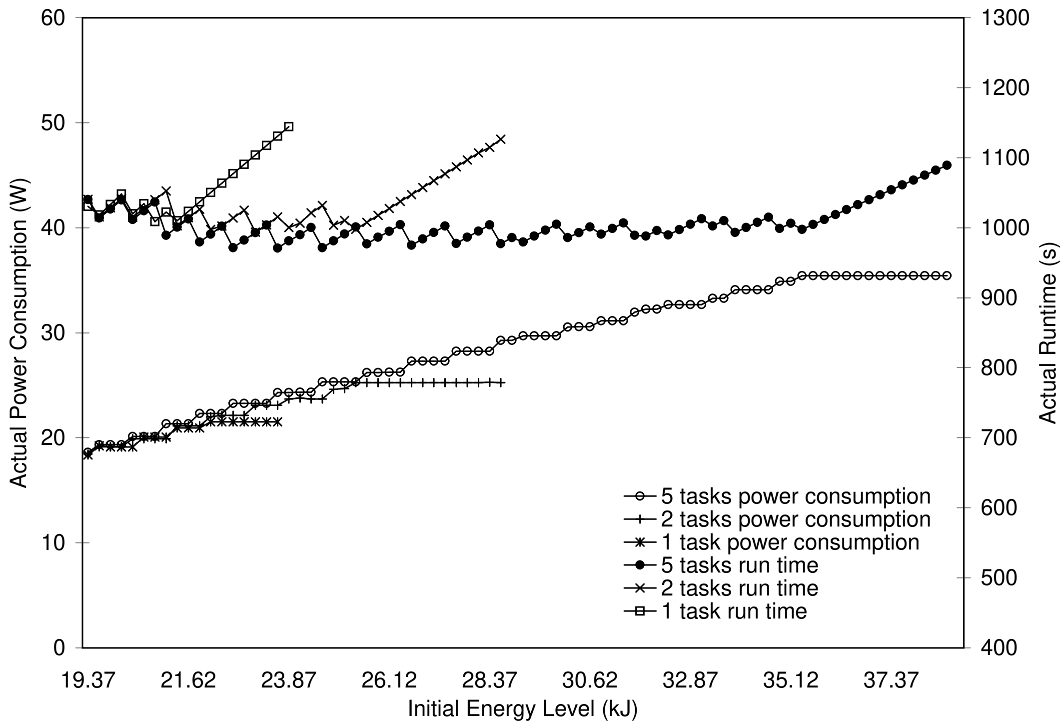


Figure 5.16: Measured power dissipation and the resulting system runtime after adaptation, no DVS.

As we do not have access to actual real-time control applications (for proprietary reasons), we create a set of real-time tasks by modifying `lame` [85], an open-source MPEG Layer-3 (MP3) audio encoder to operate as a periodic real-time task on top of our EQoS framework. Adaptation is performed by varying the "quality" parameter, which selects psychoacoustic models of varying complexity, resulting in a tradeoff between output quality and processing time/energy. The real-time and power characteristics of this task for various QoS levels are shown in Table 5.4. Note that at the lowest service level, the task is simply not run, and that the utility values were selected such that they provide decreasing marginal returns for each higher QoS level.

We ran the system using the greedy adaptation heuristic, with a target runtime, t_{run}, of 1000 seconds. The task sets consist of multiple instances of our adaptive `rt-lame` task. We vary the total energy parameter, set the power budget assuming 17 W fixed draw, perform adaptation, and measure the power dissipation of the laptop. Figure 5.16 shows the power dissipation after adaptation for task sets with 1, 2, and 5 instances of `rt-lame` and DVS disabled. Also shown is the resulting runtime based on the input energy parameter. As

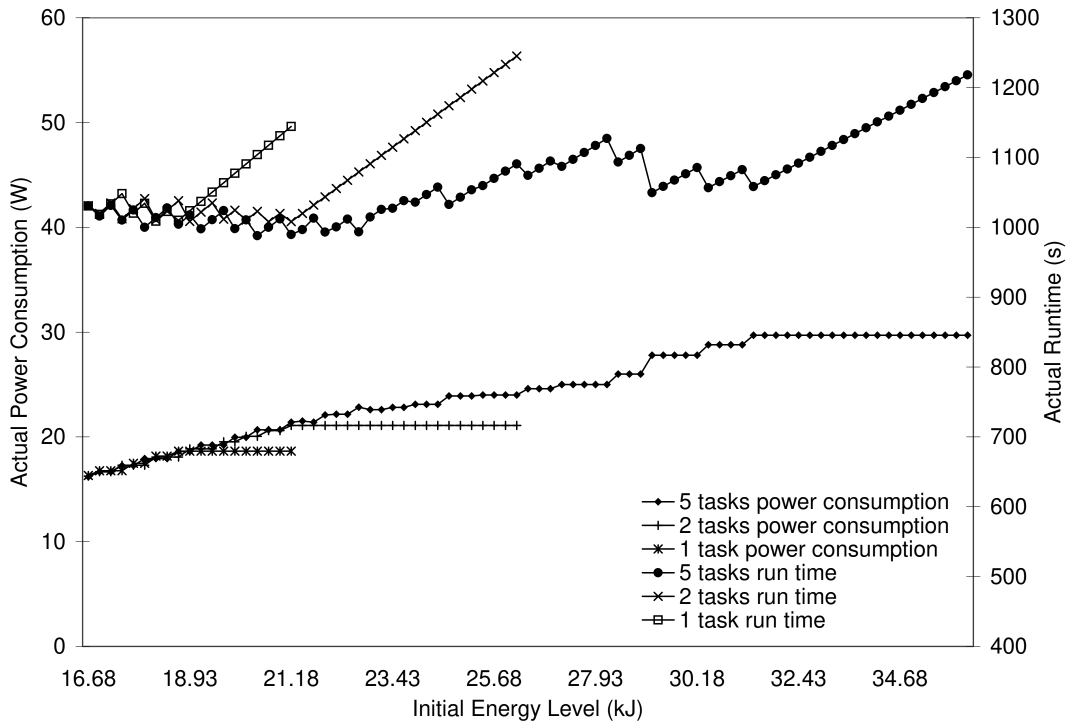


Figure 5.17: Measured power dissipation and the resulting system runtime after adaptation, with DVS and compensation.

we can see, when energy is constrained, all three cases closely achieve the desired runtime of 1000 s. Once there is sufficient energy, of course, all tasks are run at the maximum service level and the system runtime increases linearly beyond the target time.

Repeating these experiments with DVS and compensation enabled for additional energy conservation, we obtain the results plotted in Figure 5.17. Here, task are executed at higher QoS levels in an attempt to keep the same power dissipation to meet the target runtime. However, in spite of the compensation, the DVS results in 5–10% lower power dissipation and reciprocal increase in runtimes. With a known time-to-charge scenario, the energy providing this extra runtime could have been better spent running tasks at higher QoS levels to provide greater utility over the target runtime. This again shows that, especially when DVS is used, it is best to adapt task sets periodically, rather than just once, to minimize deviation from the target runtime.

Finally, in Figure 5.18, we show the utility gained from the completed computation of the system over the target 1000 seconds of runtime. The task set is adapted with available energy, and total utility increases stepwise, indicating the transition between particular

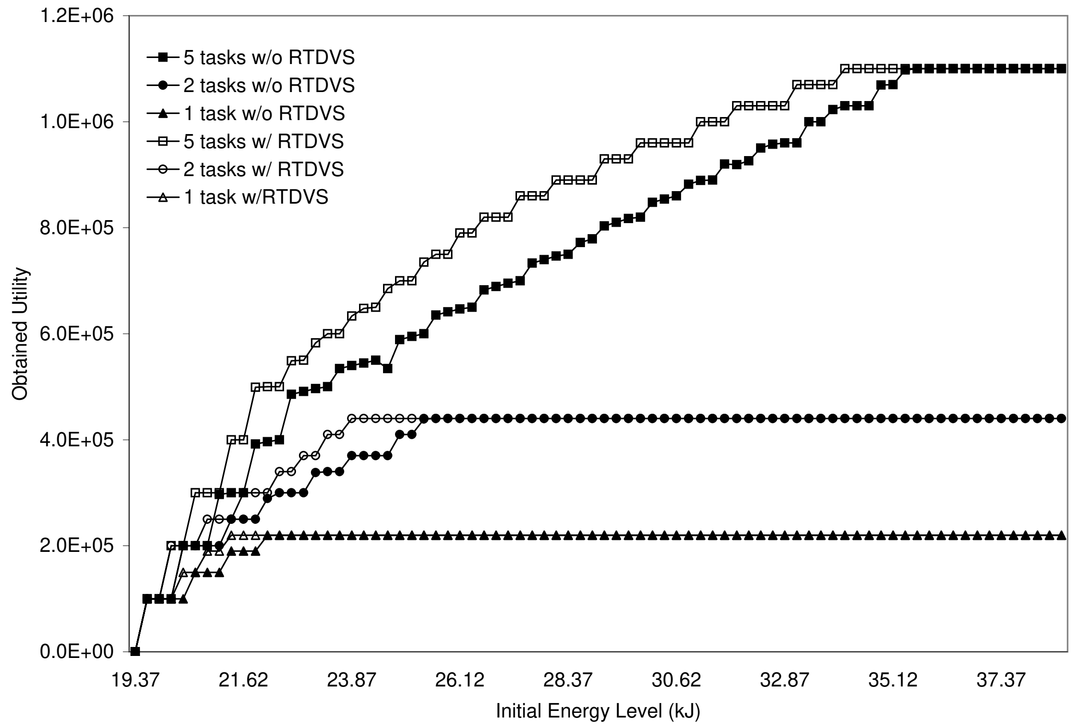


Figure 5.18: Resulting total utility until 1000s, with adaptation, with and without DVS.

combinations of QoS levels, until all tasks are at their maximum service levels. In these particular task sets, with this laptop's voltage-scaling capability, using DVS provides up to 15% improvement in utility. Of course, this depends on the utilities assigned (e.g., whether marginal utility gains are decreasing or increasing when running tasks at higher QoS levels) and the actual voltage scaling improvements possible on the specific hardware.

5.6 Related Work

Energy is becoming an increasingly common objective in system optimization. Much of recent research has focused on energy-conserving techniques, especially on the use of DVS. Since the earliest work by Weiser *et al.* [90] on voltage scaling, several papers [23, 59, 60, 62, 69] have dealt with DVS for energy savings in general-purpose machines by scaling frequency and voltage to the detected processor load/idle time, or by stretching execution to some target time. More recently, some have used prediction of episodic interaction [20] or soft deadlines and task workload estimation [47] to maintain good human-interaction and multimedia performance with DVS. Although some of the earlier works claim real-time capability, they are not sufficiently rigorous to work with the canonical model of periodic real-time tasks, and, therefore, are not directly applicable to our real-time EQoS framework.

In the domain of real-time applications, it is more difficult to apply DVS, as it is not easy to provide scheduling guarantees in the presence of changing frequencies and worst-case execution times. A few recent works [24, 36, 56, 65, 82] have managed to provide DVS in a true real-time context. Generally, they combine offline analysis with some form of online reclamation or slack-stealing [42] mechanism to ensure deadline guarantees while minimizing energy consumption. Practical heuristics, such as those implemented in [65], are an important element in our EQoS framework.

Various approaches to application adaptation [39, 58] in resource-constrained environments exist in the current literature. A few projects are targeted more specifically to energy adaptation. The Milly Watt project [18] explores application involvement in energy and power management in PDA-class devices. Flinn and Satyanarayanan [21] adapt multimedia applications to ensure a user-specified runtime on battery-powered laptops. Our work extends to real-time systems, where we are restricted by timeliness guarantees and worst-case execution limitations, but have the advantage of generally well-specified task sets.

In real-time systems, adaptation has mostly been restricted to the fault-tolerance domain. Generally, adaptation results in graceful degradation through period extension [75] or by eliminating occasional invocations of tasks [71]. Other works provide reduced service using imprecise computation models [13, 46, 77], and primarily focus on avoiding overload in fault-tolerant, multiprocessor systems [10]. Our EQoS framework leverages these fault-tolerant service degradation techniques and applies them to reduce computational and energy requirements in single processor embedded systems.

In a real-time context, one recent paper [73] addresses value maximization subject to energy constraints. However, the model assumed is much more restricted, using tasks with a common deadline and selecting a subset of the tasks to maximize value, rather than the more general problem of varying QoS to an unrestricted set of real-time tasks. One advantage of the restricted model is that voltage scaling decisions are directly accounted for, rather than compensated later, though the paper does not address random effects of task execution time or optimizing for actual system runtime.

We have formulated the basic energy adaptation problem as a selection of QoS levels.

This is reducible to a 0-1 multiple-choice knapsack problem [52], a lesser-known variant of the 0-1 knapsack problem. A few optimal solutions [17, 52, 57, 79] exist for this problem, as well as for its linear relaxation [67, 92]. We use simple optimal algorithms [52] and approximation heuristics for solving MCKP in our EQoS framework.

5.7 Conclusions

In this chapter, we have developed an EQoS framework that provides adaptation of task sets in energy-constrained embedded real-time systems. By leveraging existing methods of real-time adaptation for fault-tolerance and graceful degradation, we have proposed a general adaptive task model and formulated the energy-adaptation problem in a tractable and solvable form. We have shown a couple of optimal solutions as well as simple heuristics to provide maximum benefits or utility with a limited energy budget and a known time-to-recharge. This solution may, in turn, be used to achieve other energy-adaptation goals, such as maximizing benefits irrespective of system runtime.

We first presented detailed simulations showing the relative performance of different adaptation algorithms in maximizing utility for a wide range of task sets. Overall, the optimal solutions outperformed the heuristics, but incurred significantly higher execution overheads. The greedy heuristic turned out to be a good compromise, achieving 0.9 of the optimal value for at least 95% of the task sets we examined. Although the inclusion of energy-conserving DVS mechanisms can greatly increase runtimes and variability, we were able to effectively account for these effects and achieve the desired runtime with much improved utility gain.

This EQoS framework has been implemented as an adaptive real-time extension to the Linux operating system. Through measurements on a laptop running a set of audio encoding applications, we have demonstrated the efficacy of energy adaptation in a real-time environment. The measured power consumption after adaptation closely matches the power budget specified, resulting in the desired runtimes and providing maximal utility.

CHAPTER 6

Self-Monitoring and Modeling of Task Energy Consumption for Power-Aware Operating Systems

A key component of an energy-aware operating system is the ability to allocate scarce energy resources to tasks in order to make the best use of the limited available energy. The previous chapter described an Energy-Aware Quality-of-Service framework that allows the operating system to perform adaptation of the work set to maximize returns on computation and energy expended. However, in order to apply such mechanisms, an accurate measure or estimate of the energy needs of each task is needed on the mobile target platform.

Due to the short time intervals (on the order of a few milliseconds) involved, and a fluctuating power draw, it is difficult to measure the energy consumed by a single task in a running system. In this chapter, we explore various measurement techniques and develop a low-cost hardware solution that addresses the drawbacks of existing techniques. Using measurements of task energy, we develop several models to characterize task energy consumption, requiring only a few platform parameters and execution characteristics to predict task energy consumption. Comparing these predictions against measurements of real applications running on laptops, we verify the accuracy of our models and show that they typically estimate task energy consumption to within 5% of actual values, and should suffice for use in energy-based task adaptation systems.

6.1 Introduction

As computer technology progresses making smaller, lighter, and faster computing devices available at inexpensive prices, we are experiencing a proliferation of handheld and mobile devices that are capable of high computational performance and sophisticated functions. The exponential improvements in processing speed and similar gains in large-scale integration allow processing power and memory capacity that a few years ago were reserved exclusively for workstation-class machines to be incorporated into handheld PDAs. However, this performance and functionality has not come for free. In particular, despite the gains in low-voltage technologies and improvements in semiconductor fabrication, energy consumption of high-performance and multi-function devices has continued to increase. To make matters worse, these increases have far outstripped any improvements in battery technology, increasing the gap between energy consumption and available storage in a reasonable size and weight mobile device. As the demand for high-performance applications, such as multimedia and gaming on PDAs, or video and web browsing on cellular phones, is likely to continue growing at a rapid pace, the market pressures to build increasingly high-performance, mobile and hand-held platforms will remain unabated. This results in a serious engineering issue to bridge the energy supply-demand gap, and provide both high performance and reasonable battery-life into a lightweight, compact form factor.

Because of these issues, power management is becoming a highly critical issue in the design and implementation of mobile computing platforms. The simplest methods of reducing energy consumption involve hardware designed to be able to shut off, or remain in a low-power standby state when not actively used. More sophisticated approaches use dynamic voltage scaling (DVS) [90] techniques that reduce the operating frequency and voltage of a processor in order to lower per-cycle energy costs when the system is underutilized. Even for embedded systems that have time-critical code, where latencies for switching out of low-power states, or increased execution times due to reduced operating frequency, can cause real-time applications to miss deadlines, such techniques can be used very effectively for reducing energy consumption, as shown in Chapters 3 and 4, and in several recent papers [24, 33, 36, 56, 65].

All such methods effectively stretch how much computation a finite battery capacity can allow, but do not address whether the energy is being well-utilized on valuable computations. To address this issue, adaptation techniques have been developed that try to allocate energy to more valuable or higher utility tasks by changing the task set or varying the quality of service provided by each task [72, 73]. The EQoS framework (Chapter 5), in particular, executes tasks at degraded quality of service levels to reduce the computational and energy demands on the system. For a given amount of stored energy, adaptation algorithms optimize the total value provided by the system by differentiating QoS of each task and selecting the highest service levels such that runtime requirements are met. The energy-aware adaptation mechansisms try to ensure maximal returns on scarce energy resources in portable devices.

However, for these techniques to be effective, detailed information about the running system is necessary. In particular, for each task in the system, one needs to specify all runtime energy-consumption characteristics for all possible service quality levels. For real-time and embedded control systems, generally task sets are already well-defined with respect to execution times of tasks and various timing constraints, so it is not a great stretch to also provide energy information in the task-set specifications. However, in more general-purpose systems, providing energy information, particularly for all possible degraded quality levels of each application, can be a significant burden.

In order to utilize advanced power management and adaptation techniques, we need comprehensive methods of providing task energy requirements on mobile platforms. In this chapter, we first look at methods that directly measure energy of tasks running on a system and evaluate cost-effective methods of obtaining energy measurements for adaptation feedback. Based on such measurements, we further propose several models to predict and estimate energy dissipation of task execution. Using just a few measured parameters on a particular hardware platform, we show that task-execution energy can be predicted with a high degree of accuracy based solely on runtime and execution characteristics.

In the following section, we will look at methods of measuring execution energy, and then design a low-cost hardware tool to measure task energy of running systems. We then develop several parametric models to predict energy consumption of tasks and discuss their applicability to various systems. After evaluating our energy models, we finish this chapter with some concluding remarks.

6.2 Measuring Task Energy

Aggressive power management strategies rely on adapting the working task set to the current energy resource and runtime constraints through quality-of-service and other degradation schemes to maximize the benefits of computation with the limited energy available. However, in order to use service adaptation algorithms, one needs well-defined task sets for which energy consumption metrics are known. In this section, we will discuss some existing methods of measuring task energy. We will then propose a cost-effective measurement device that can provide accurate task-granular energy measures at runtime for both adaptation and performance feedback.

6.2.1 Energy Measurement Methods

To take advantage of advanced adaptation algorithms that can improve the utility gained from execution with limited battery capacities, we need to provide accurate energy consumption profiles for all of the applications in the system. Typically, the task energy is expressed as the average energy per invocation or per scheduling time slice, depending on the type of scheduling used in the system. The most direct method of determining task energy is to actually measure the power consumption of the target platform when executing these tasks. However, as the actual execution patterns and corresponding energy consumption can vary, and as the measurement intervals are very short when compared to human-perceivable time scales, measuring task energy involves some difficult issues and obstacles.

The simplest method of measuring power consumption is to measure the current flowing into the device while executing a particular task. Figure 6.1 illustrates this setup, using a digital multimeter to measure the current and supply voltage. The energy dissipated is simply a product of the supply current, supply voltage, and execution time of the task. This assumes that the power dissipation over the entire execution time is constant. Unfortunately, this is not the case, as the true power consumption can vary based on various factors, including the types of instructions currently executing, pipeline stalls or flushes, and memory access on cache misses, so the power consumption will vary continuously while executing a task. In addition, the sampling time on multimeters is generally very

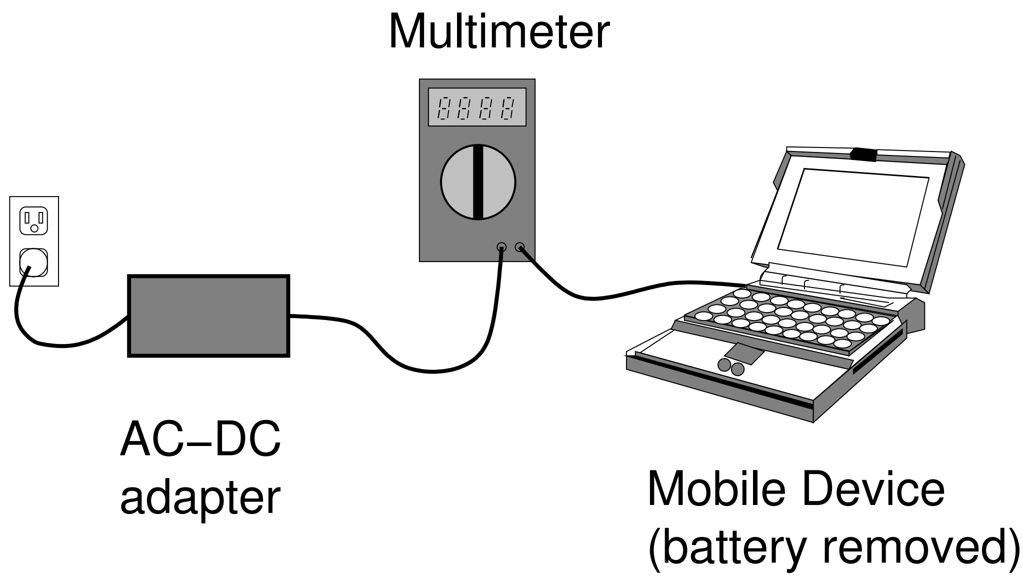


Figure 6.1: Power measurement using a multimeter.

long compared to the millisecond-range execution times typical of a single time slice of execution, so we cannot hope to get more than one current measure for a particular invocation of a task. Furthermore, it is not always clear whether these are instantaneous measures taken with sample-and-hold circuits, or if some sort of averaging is occurring over the sampling interval. For the former, the point measures may not reflect the actual consumption over the entire time slice, while for the latter, due to long sampling intervals, averaging will include time well beyond the time slice in which the task executes. Finally, there is also a problem in ensuring that the measurements are made synchronously with the actual execution of the task in which we are interested, rather than at some arbitrary point in time when something else may be executing.

To address some of these issues, Flinn, *et. al*, [22] have developed the PowerScope tool. Basically, this uses a multimeter that takes point measures of current, but also generates a trigger output when samples are taken. By connecting this trigger as an interrupt source to the target platform, they can determine and log the exact point within the task execution to which each measurement corresponds. Over many repeated invocations of the task, they can statistically build up a profile of the execution energy of the task by analyzing the logged measurements. Although this does work in measuring task energy, it is quite intrusive, needing additional interrupts, and is not easy to actually perform, requiring a

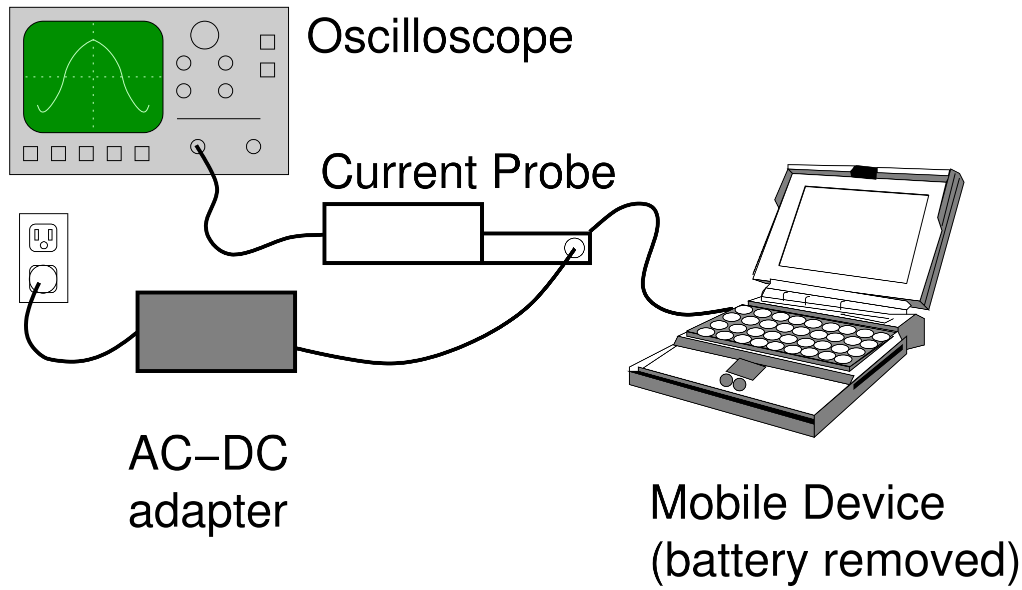


Figure 6.2: Power measurement with an oscilloscope.

fair amount of technical expertise just to take a few measurements. It may be possible for application developers to measure and provide such energy measures, but this method requires a significant time investment for each task energy measurement, and it does not lend itself to automatic self-measurement in the actual mobile platform.

An alternative approach overcomes the slow sampling rate of multimeters by using an oscilloscope. When equipped with current probes, as shown in Figure 6.2, an oscilloscope can show detailed, high frequency power consumption information. With the high sampling rate, it is no longer necessary to continuously repeat the task and take multiple point measurements to statistically generate a profile — a straightforward measurement provides the energy profile for a single invocation of the task. Synchronizing the measurements with the task execution is fairly simple — one can make the target platform toggle some I/O pin when starting the time-slice for the task, and use this as a trigger for the oscilloscope. This is far less intrusive than generating an interrupt on the device and logging information there. Even electrically, this method may be less intrusive, as oscilloscope current probes often use Hall-effect sensors, and do not need to be connected in-line with the power supply. The energy consumed by the task is simply computed by integrating over the measured current profile and multiplying by the source voltage. Some digital oscilloscopes can perform these computations themselves, and also interface to a monitoring computer, making the measurement process even simpler. However, this approach requires a substantial investment in equipment that may not be available to end users, and cannot be implemented within the mobile device for self-monitoring.

The only alternative is to use custom hardware to make the task energy measurements. The major drawback of custom hardware is that it is often expensive to implement and test. In the following section, we present our design for a low-cost task energy measurement device that can readily interface to mobile platforms for self monitoring and adaptation feedback.

6.2.2 Charge-Flow Metering

In order to use advanced quality-of-service adaptation algorithms to best utilize the energy on mobile platforms, we need a low-cost, simple method of measuring and specifying task energy consumption. We would like to develop a small hardware device that fulfills the following requirements:

1. it is very inexpensive to implement,
2. it measures energy consumption over time intervals comparable to a scheduling time-slice, and
3. it can readily interface with mobile and handheld platforms.

The ability to interface with the platform being measured is needed to synchronize measurements with the actual execution of tasks. Additionally, by using some form of digital output, we can automate measurements and allow the mobile platform to perform self-monitoring of energy expenditures. The simplest interface likely to be available on such platforms is an RS-232 serial port, although these are rapidly being replaced with smaller form factor, higher speed USB ports on newer consumer devices. The serial interface is ideal for high level control and transferring measurement data, but as it requires on the order of 1 ms to transfer a byte, it is too slow to use as a trigger to synchronize the start of measurements to the execution of a task. For this purpose, we will use a dedicated output pin from the measured platform, such as the DTR or RTS control pins on the serial port, which can be toggled very quickly, on the order of 1 μs.

To provide digital measurements, some form of analog-to-digital conversion needs to be performed. The cost of an A/D converter can vary greatly, depending on the precision, expressed as the bit-width of each sample, as well as the speed, which defines the maximum sampling rate. A converter that can provide measurements comparable to an oscilloscope will require a very fast, very expensive device with a large bit width. On the other hand, a similarly precise converter that operates at much lower speeds (few hundred to few thousand samples per second) are fairly inexpensive and should suffice for our purposes. However, we need to be careful to avoid measuring current at such low sampling rates, or else we will run into the same issues as with multimeter-based task energy measurements discussed in Section 6.2.1.

Instead of directly measuring the current flowing into the computing device, we propose measuring the total charge that has passed through the device over the interval that the task executes. Our design for this is based around the Maxim MAX471 [53], a low-cost integrated circuit that implements a supply-side sense amplifier. When connected in series with the power supply, this device produces current on an output pin proportional to the supply current draw, at an approximately 1:2000 ratio. The output of this device is intended to be connected to a resistive load, so the resulting voltage will be be directly proportional to the current. Instead, we use the output to charge a capacitor, as shown in Figure 6.3. The capacitor acts as an analog integrator, summing the current on the output pin of the MAX471 over time. As the voltage across the capacitor is proportional to the total charge stored ($V = CQ$), which, in turn, is directly proportional to the total charge that has flowed from the power supply, the measured voltage across the capacitor indicates the total charge that has been consumed by the computing device.

To measure the energy used in executing a particular task, we first ensure the capacitor is discharged using a transistor to short it to ground. At the beginning of a time-slice for the task, we turn off the transistor to allow the capacitor to charge. When the task completes its invocation, we measure the instantaneous voltage on the capacitor. As the MAX471 and the charging capacitor both exhibit simple linear behavior, we can multiply this measured voltage by a calibration constant to determine the total charge consumed during the execution of the task. Multiplying this by the supply voltage gives us the energy consumed executing this task.

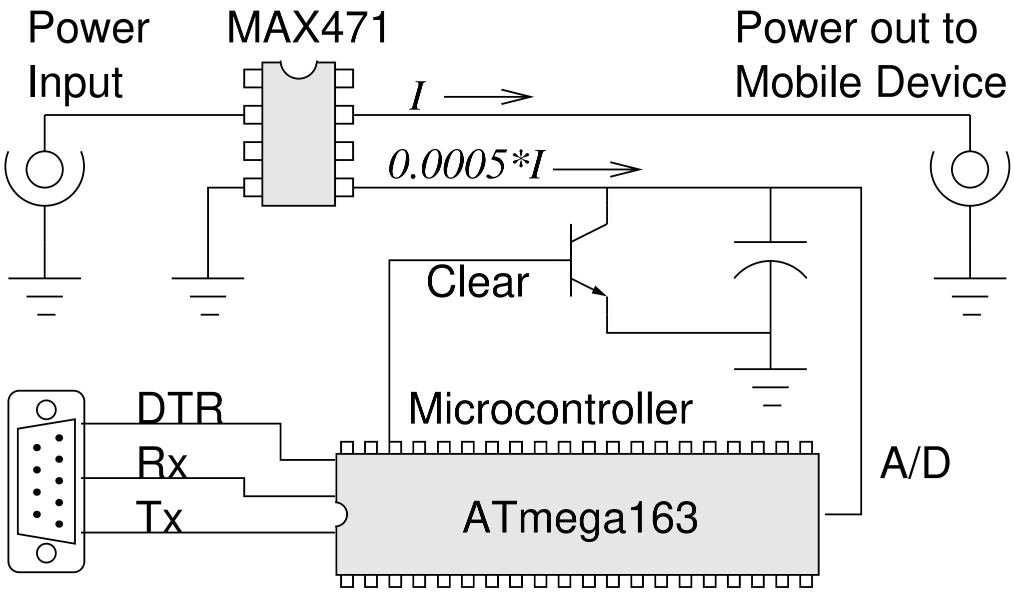


Figure 6.3: Charge flow measurement hardware.

Due to the fast response time of the sense amplifier (less than 1 μs), even high frequency changes in current consumption are accounted for in the final charge on the capacitor, so we do not need a very fast A/D converter when this method of measuring charge is used. A low-cost converter, such as those that are integrated with microcontrollers, suffices. Since there are some specific control functions that need to be done (e.g., sensing trigger, discharging the capacitor), as well as some data path functions (e.g., adjust for calibration, forward on serial port), we can make good use of a small, microcontroller with built-in A/D converter. We based our design on the Atmel AVR series 8-bit system-on-chip type microcontroller [4]. These have internal clock generators, flash/RAM/EEPROM memories, multichannel 10-bit A/D converters with internal reference, and serial ports on a single IC. The only external parts needed are the MAX471 sense amplifier, the capacitor and transistor for charge accumulation, a serial driver to match RS-232 voltage levels, and some method of providing a 5 V power supply.

To maximize the accuracy of measurement over our desired time granularity, an appropriate value for the charge accumulating capacitor must be selected. The internal A/D reference voltage is nominally 2.56 V, so we need to ensure the capacitor stays below this voltage over the task execution intervals. We designed this circuit to measure the energy consumption in a PC laptop computer, that draws up to approximately 2 A from an external 19.5 V power supply when not charging a battery. As the MAX471 produces an output current approximately 1/2000th of the supply draw, by using a 22 μF capacitor, we can expect precise measurements over a range of 1–50 ms, corresponding nicely to the typical scheduling time-slice for task execution.

All of the values specified for the parts are nominal values, so the actual characteristics may be significantly different. In particular, up to 20% error may be expected for the capacitor. So we must calibrate our device against some other form of energy measurement. We note that all of the relevant circuitry, in particular the MAX471, the capacitor, and the A/D hardware, are highly linear in response, so only a single multiplicative constant is needed to convert the measured values to absolute energy units. Using a constant resistive load and a fixed time interval for measurement, we can calibrate the device measurement against the power computed using an ammeter current reading. This calibration value is then stored in the internal EEPROM of the microcontroller for future use. This single calibration should be effective for all future measurements as long as the temperature is not significantly different from that at calibration.

This implementation is effective at measuring the total energy consumed over a time interval corresponding to a task invocation or scheduling time-slice. The device reacts very quickly to the trigger pin, starting the measurement within 20 μs of the trigger edge. It has similar reaction time to the trigger to stop measurement, but due to the sample-and-hold timing, requires an additional 6 μs. This error in the measurement window should account for less than one percent error in the final energy values. In addition, by investing in a fast external clock circuit, these latencies can be reduced 8-fold. The actual A/D conversion, adjustment for calibration, and subsequent transfer of data over the serial port incur the greatest latencies, on the order of 15 ms, so it is this time that limits the maximum rate of energy measurements that can be taken with our design. Since we use only about one quarter of the RAM available on the microcontroller, and since we have already parallelized the actual measurement phase with the computations and serial output in the microcontroller software, a slight upgrade to this software to buffer measurements on-chip and send them out later will allow bursts of up to a few hundred consecutive measurements that are spaced only 100 μs apart.

With this low-cost hardware design, we have a very easy-to-use method of determining task energy consumption, requiring only a trigger output from the measured platform to synchronize the measurements with the task execution, and a serial interface on either the target platform or on some logging machine to record the energy measures. As the interfacing and measurement methodology is very simple, it is not difficult to automate the process and use this hardware to allow the measured platform to perform self-monitoring of energy consumption and provide feedback to energy-based task adaptation mechanisms.

6.2.3 A Self-Measurement Architecture

Using the type of energy measurement device designed above, it is a fairly straightforward process to create a self-monitoring platform that continuously updates energy consumption profiles of applications, improving the accuracy of energy adaptation algorithms. A simple software architecture for performing self-monitoring, built on top of Linux, is described below and illustrated in Figure 6.4.

We assume that some form of energy-based task-adaptation system has been implemented in the kernel or as a middleware layer. This system adjusts the quality of service provided to several computation-intensive tasks that run in user mode. There is also either a user-level daemon or a kernel interface to provide descriptions of the tasks, in particular, the amount of energy these require for each invocation or time-slice. The self-measurement system consists of a user-level daemon that performs most of the functions, and a small kernel module that triggers measurements.

Our self-measurement daemon interacts with the task description interface of the adaptation subsystem. We iterate through the list of tasks, selecting one at a time for measurement. The small kernel module is attached to a hook inside the scheduler and sets an output pin (e.g., DTR signal on the serial port) to high when the target process is scheduled for execution. Similarly, when some other task is scheduled, the pin is set to low to trigger the end of the measurement. The only tricky issue is that when stopping a measurement, the module keeps a time stamp to ensure that a subsequent measurement is not started too soon (see latency discussion at the end of Section 6.2.2).

The user-level daemon receives the measurements over the serial interface, and then updates the target task's energy consumption value for the current level of service provided

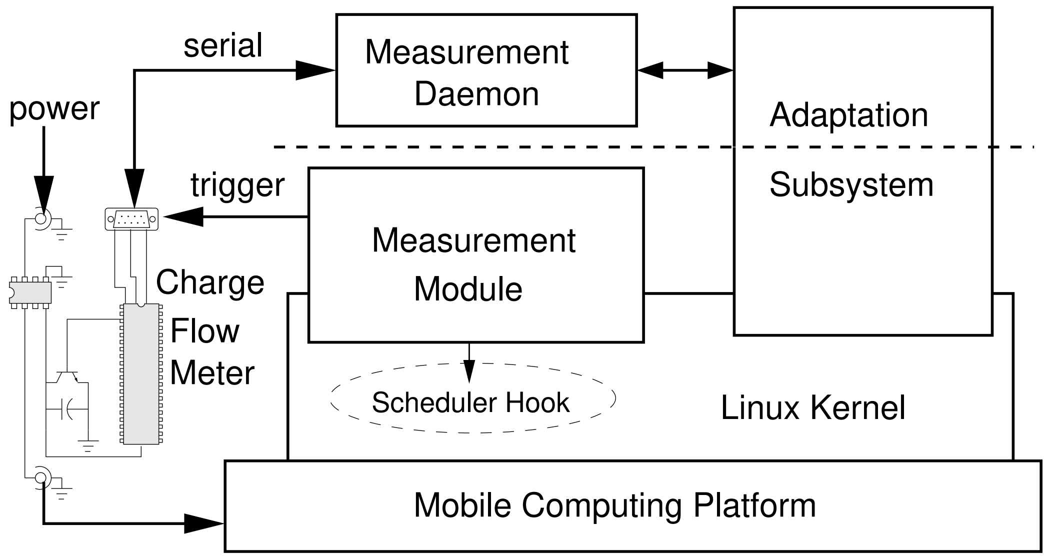


Figure 6.4: Self-measurement software architecture.

by the adaptation subsystem. This update should be a form of time-average filter, but the design of such filter will require a trade-off between agility and stability [32]. By cycling through the tasks of interest, all task energy consumption values will gradually be updated to reflect any changes dynamically.

Of course, the tasks will receive updates only for the service levels that are actually executed. For the service levels that are not selected by the adaptation system, the task energy metrics will not be updated. To ensure that all tasks do receive updates for all possible service levels, we would need a method by which the adaptation module can be bypassed, so tasks can occasionally be executed at service levels deemed non-optimal, and the corresponding energy consumption specifications can be updated. The actual mechanism used to do this will depend heavily on the implementation of the adaptation system. As this bypass mechanism will be invoked very infrequently, it will not significantly affect the functioning of the adaptation system.

This architecture, combined with the low-cost charge flow meter discussed earlier, can allow a mobile or handheld device to monitor and adjust the parameters of QoS adaptation to the actual energy consumption of the executing tasks. The overhead for this measurement is very low, since the additional hook in the scheduler requires only a few instructions, and the module can be set to make infrequent measurements. The user-level daemon stays blocked between measurements, while the external hardware described above, stays in a low-power sleep mode, mitigating any additional computational or energy overheads.

6.3 Modeling Task Energy

In order to use advanced task-adaptation techniques on mobile platforms, it is necessary to provide an estimate of task energy consumption. We have so far seen how one can measure the energy consumed in executing a task on a particular platform, and how to automate the acquisition of task energy information on the target device itself. However, this does require additional hardware and, despite the low-cost, it may not be desirable to add this to a consumer device. As it is generally unreasonable to expect the end users to perform energy measurements to optimize the battery-life of the system, the onus of measuring task energy would fall to the application developers. Unfortunately, the sheer number of different platforms on which an application needs to run may make measurement impractical. Even when restricting the class of machine by OS, such as PocketPC, and further restricting by processor, such as ARM, there are still multiple platform vendors, supplying several different models, each of which is unique from an energy perspective, due to differences in the actual hardware components used.

In this context, directly measuring task energy consumption may not be practical. Instead, we would like to find some method of estimating the energy consumption of the tasks. In particular, it would be more practical to distill characteristics of a task's execution, and using this along with some parameters describing a given platform, predict the execution energy of this task on this platform. To do this, we will identify useful traits and develop models that estimate task execution energy.

6.3.1 Constant Power Models

We first simplify the problem of estimating task energy consumption by eliminating all of the possible variables that can affect the energy requirements, and rather focus on only the most significant trait. We believe that the single most significant characteristic determining the platform energy consumption is the total execution time. Furthermore, the longer this execution time, the greater the energy consumption. Intuitively, this seems to be a simple linear relationship, so we can model platform energy consumption as simply

$E = C * t$, where t is the execution time, and C is a platform-specific constant determined empirically. This model, therefore, assumes a constant power dissipation by the platform hardware.

Unfortunately, this simple model does not capture the variability observed in measurements of energy dissipation. This model predicts behavior similar to a resistive load, while in practice, we see continuous variation in the power dissipated. Furthermore, this model is not particularly useful for any form of power management, let alone task adaptation, as it assumes constant power regardless of the computational workload, and will therefore predict a constant battery runtime for a given starting energy regardless of any adaptation.

6.3.2 Bimodal Power Model

We can slightly extend the constant power model to be both more accurate and more useful for adaptation. We note that when idle, the processor dissipates very little power, due to aggressive halting of the core in place of wasteful idle loops, but it is a major consumer of energy when tasks are actively running. Therefore, we can construct a better model of platform power dissipation as a bimodal constant power system. We assume two states for the system, active and idle. When the processor is idle, the model assumes a constant power dissipation, P_i. Similarly, when executing a task, a larger constant power, P_a, is assumed. As before, a task that executes for time t will expend energy $E = P_a * t$. With n tasks, the total system energy consumption is $E = P_a \sum_{i=0}^{n} t_i + P_i t_{idle}$, where t_i is the total execution time for task i, and t_{idle} is the total idle time. Therefore, this simple bimodal constant-power model is useful for adaptation — although task energy is solely determined by execution time, system energy consumption depends on the processor utilization. Decreasing the workload and increasing idle time will reduce the system energy consumption.

Using a simple loop that executes a stream of integer operations, we can determine the constants for a given platform. We use a PC notebook computer, HP N3350 based on the AMD-K6 processor [2], and our task energy measurement hardware described earlier. By measuring the energy dissipation over random time intervals both while running this loop and while the processor is halted, we obtain the results plotted in Figure 6.5. The plotted points show a very linear relationship between execution time and energy, as we expected. Plotting the best-fit lines, we can obtain the power constants that correspond to the slopes

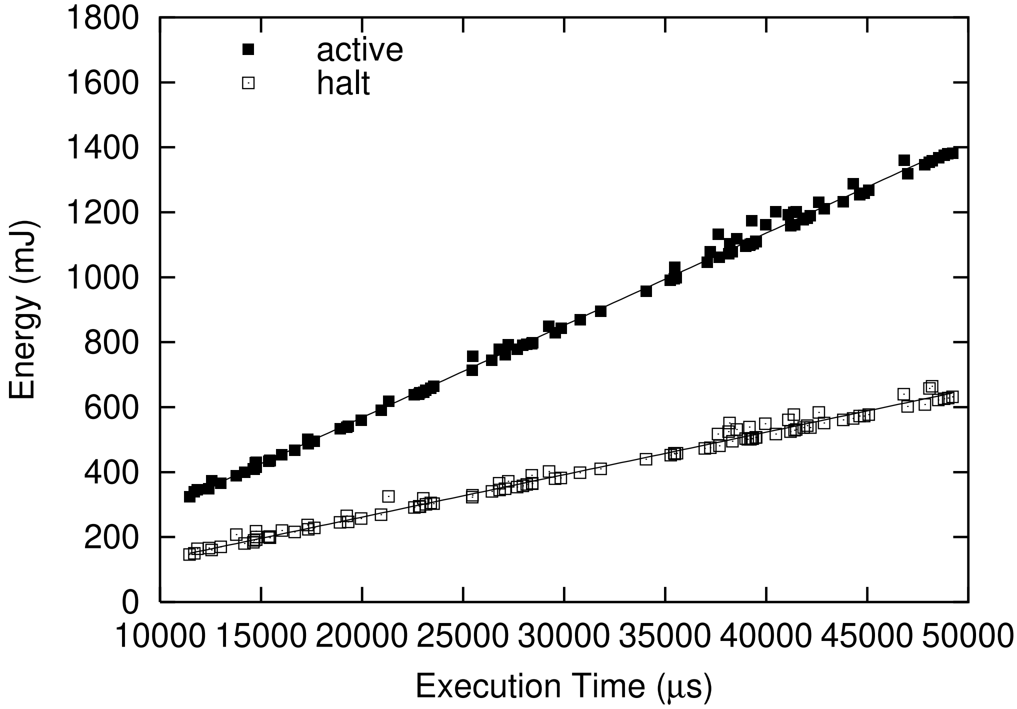


Figure 6.5: Energy measurements on HP N3350 laptop. Linear regression lines are also plotted.

of the lines. Here, we find $P_a = 28.4$ W, while $P_i = 13.1$ W.

An alternative way of viewing this model is to separate power into static and dynamic components. The system always dissipates at least the static or fixed power, P_{sys}. When tasks are executing, an additional constant power, P_{cpu}, is also dissipated, reflecting the additional energy consumed when the CPU is active. These values are simply obtained from the above: $P_{sys} = P_i$ and $P_{cpu} = P_a - P_i$. We can now specify the additional energy $E = P_{cpu} * t_i$ that each task i consumes beyond the system dissipation over time t_i. This way of specifying task energy may be more useful than the total energy, since adaptation algorithms are concerned primarily with the additional energy required to execute a task, above and beyond what the idle system consumes.

In addition, this second way of specifying the power constants separates out the dynamic component of processing energy, so any simple power management mechanisms, such as turning off a peripheral, should only affect P_{sys}, leaving the task energy specified by P_{cpu} untouched. We can see this in our measurements of the laptop. Based on the

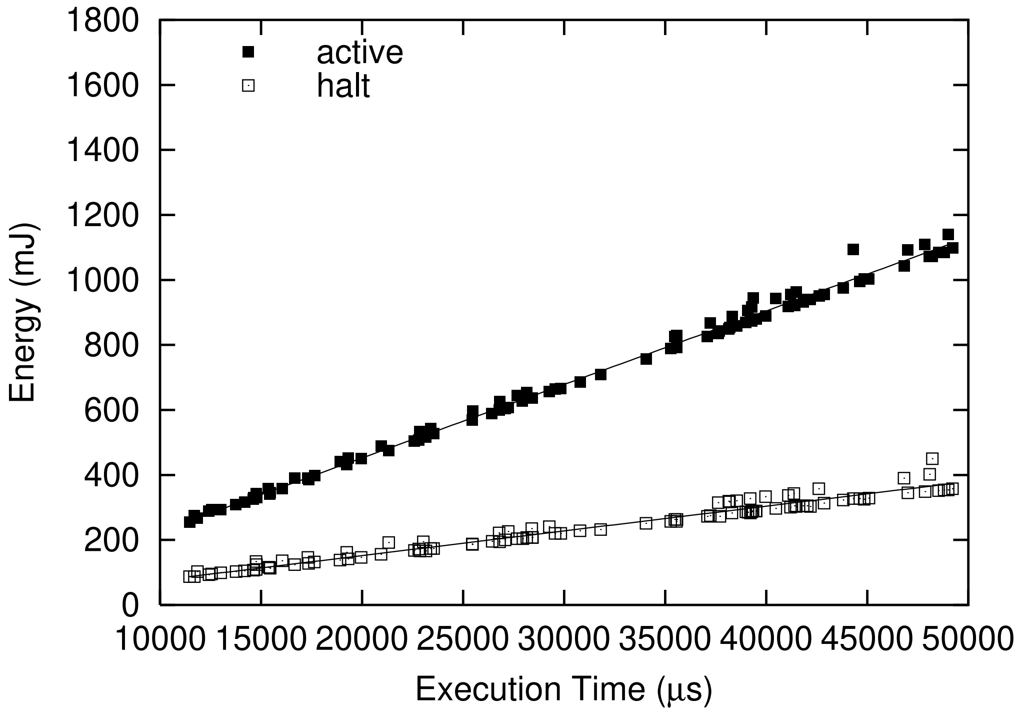


Figure 6.6: Energy measurements with display backlight turned off.

measurements in Figure 6.5, P_{sys} = 13.1 W and P_{cpu} = 15.3 W. Figure 6.6 shows energy measurements for the same conditions as before, but this time with the laptop screen backlight turned off. Based on the linear regression, we have $P_{sys} = P_i$ = 7.6 W, and P_a = 22.6 W, so P_{cpu} = 15.0 W. Hence, the effects of turning off the backlight are essentially entirely confined to P_{sys}, and adaptation based on P_{cpu} task energy specifications will not need new energy measures if this type of simple power management of a peripheral device is employed.

6.3.3 Modeling DVS Effects

We have described a simple model to estimate task energy consumption, breaking power into a static system component and a dynamic execution-dependent component. We have seen that simple management of external devices will affect the system component, without affecting the processing power component. However, this execution energy will be affected by any power management that directly apply to the processor itself. In particular, dynamic voltage scaling (DVS) techniques will profoundly affect the processing energy.

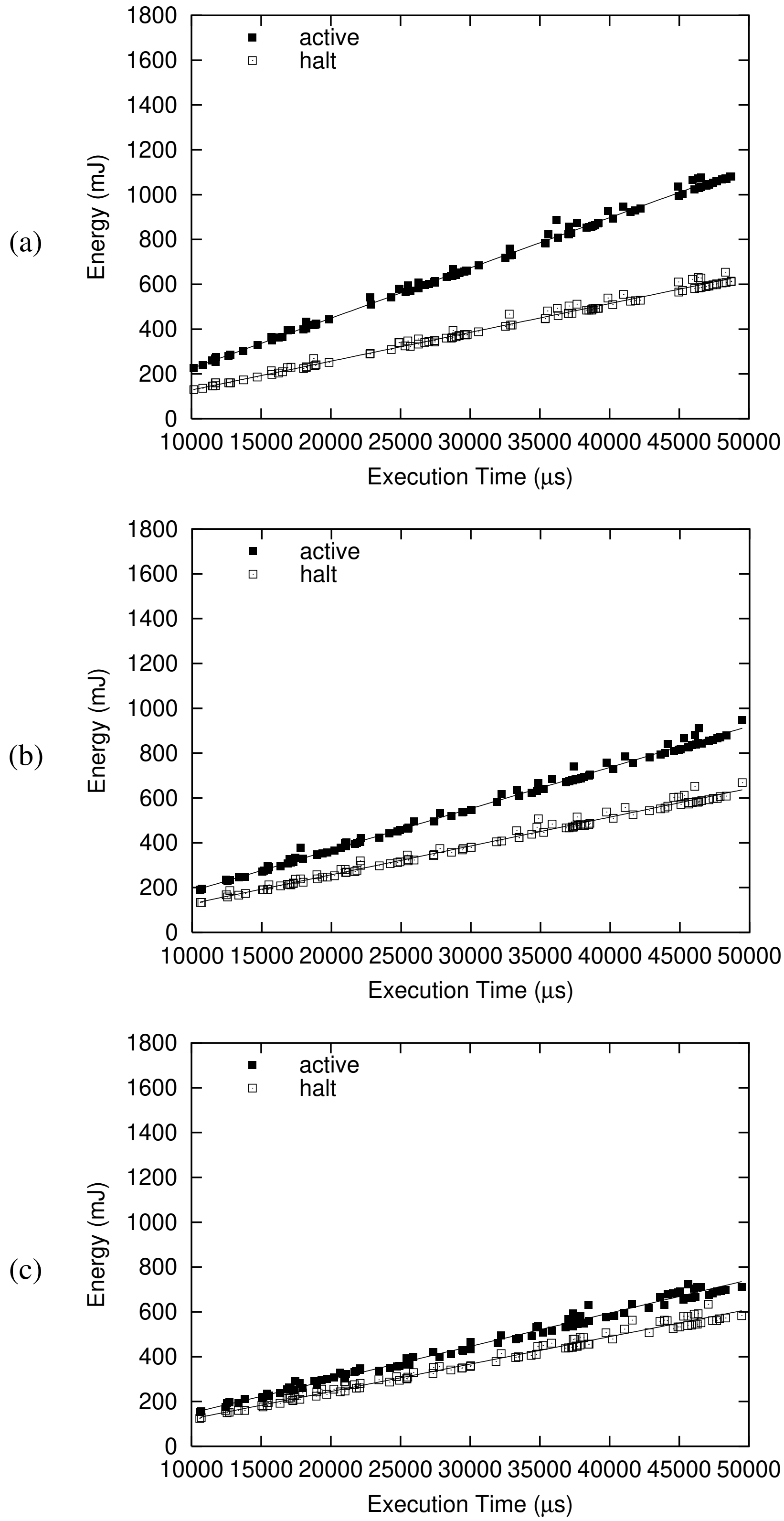


Figure 6.7: Energy measurements when frequency and voltage scaling are employed: (a) 350 MHz, 2.0 V; (b) 200 MHz, 2.0 V; (c) 200 MHz, 1.4 V.

When employing DVS, we note that there are actually two concurrent power-conserving techniques in operation. First, the frequency of the processor is changed. This directly affects the rate of instruction execution, which we would expect to reduce linearly with the frequency. This is actually an approximation, since in most systems the processor runs at much higher frequencies than the main memory, so if memory is limiting performance, the processing rate will decrease sub-linearly compared to frequency. As most processors are composed primarily of static CMOS logic, we would expect the power dissipation to reduce linearly with the processing rate.

Assuming that the dynamic component of energy primarily consists of CPU consumption, we can model P_{cpu} as directly proportional to frequency. P_{sys} should not be affected at all. Based on this we should see $P_{cpu} = P_{cpu-max} * (f/f_{max})$. We can validate this on our laptop, which supports both voltage and frequency scaling. We note that in the earlier measurements, the processor was running at 550 MHz, with a 2.0 V supply. In the first two plots in Figure 6.7, we show measurements taken with processor running at 350 and 200 MHz. Based on the best-fit lines, the P_{cpu} values are 9.6 W and 5.5 W, respectively. Using the frequency scaling model, we would expect P_{cpu} to be $15.3 * (350/550)$ = 9.7 W and $15.3 * (200/550)$ = 5.56 W, corresponding very closely to the measurements. Also, P_{sys}, measured to be 12.8 W and 12.9 W, respectively, are not significantly changed, just as expected.

The second technique applied in DVS involves changing the voltage supplied to the processor. Again, based on the fact that the processor is primarily made of static CMOS logic, its primary power dissipation is due to charging and discharging of the input gates, which act as capacitors. As the energy stored in a capacitor is proportional to the voltage squared ($E = CV^2/2$), we expect a quadratic relationship between the voltage reduction and energy reduction.

Again, assuming the dynamic component of power is dominated by the CPU energy consumption, we can model $P_{cpu} = P_{cpu-max} * (V/V_{max})^2$, while P_{sys} should remain unaffected. The last plot in Figure 6.7 shows measurements at 200 MHz, but with the processor voltage reduced from 2.0 V to 1.4 V. The measurements indicate a P_{cpu} value of 2.6 W, which compares favorably with the models prediction of $5.5 * (1.4/2.0)^2$ = 2.7 W based on the power at 200 MHz, 2.0 V. The measured P_{sys} is somewhat lower, however, at 12.2 W. This may be attributed to reduced fixed dissipation in the switching voltage regulators when producing the lower voltage output.

We can combine both effects, to produce a DVS model that predicts $P_{cpu} = P_{cpu-max} * (f/f_{max}) * (V/V_{max})^2$. Based on this relationship and the measurements at 550 MHz, we predict $P_{cpu} = 15.3 * (200/550) * (1.4/2.0)^2$ = 2.7 W, close to the observed 2.6 W for 200 MHz, 1.4 V operation.

6.3.4 Instruction Mix Power Model

Thus far, we have only used a simple loop consisting of a long chain of integer operations as the "task" whose execution energy is measured. However, the energy of a real task will differ somewhat, depending on the actual computations performed. To account for this, we extend our task energy model by determining how energy consumption varies with different instruction types.

In order to evaluate the energy consumption for individual instruction types, we need to create test tasks that are composed almost entirely of the single instruction type. For example to measure the add instruction, we can create an infinite loop that consists of a large number, say 100, of register to register additions and a single branch instruction. The power dissipation measured when executing this loop will then primarily be dictated by the instruction that we wish to test.

Of course, it is not practical to test all possible processor instructions. This is further complicated by the large number of addressing modes on architectures such as x86 or 680x0, which may allow address computation, load, and store operations within a single instruction. Instead, we need to categorize the most common types of instructions and determine a representative measure of power for each category, simplifying further by not including addressing modes. Of course, the choice of categories depends on the actual instructions available on an architecture.

For our x86-based laptop, we categorize the instructions into integer, floating point, memory load/store, and branch operations. We measure addition as representative of integer operations, but also show results for multiplication. For memory operations, we measure the energy of a loop of load instructions for which we guarantee a cache hit. For comparison, we also use a loop of memory loads and address computations that ensure

Category	Instructions	P_{cpu}
Integer	add	15.3 W
	multiply	16.3 W
Floating Point	mixed	17.2 W
Memory	cached load	17.3 W
	non-cached load	15.6 W
Branch	jump	15.5 W
	spin loop	14.6 W
Misc.	nop	15.1 W

Table 6.1: Power measurements for various instruction types.

each load incurs a cache miss. The branch is tested using a sequence of jump instructions that simply branch to the next consecutive instruction, as well as a tight, empty spin loop. For comparison, we also measure a loop of nop instructions that do nothing other than fetch instructions from the cache. The results of measuring these instructions is presented in Table 6.1.

Overall, for this platform, power varies over a range of more than 2 W depending on instruction type. We note that the spin loop results are abnormally low compared to the other results. This is due to the fact that it is the only test case that does not involve a long sequence of instructions inside a loop, and instead, has a single instruction that branches to itself. As this is the only instruction that is executed, the lower power may be primarily due to reduced activity on the processor's instruction cache. Cached data loads, similarly, increase power dissipation over non-cached loads, due to data cache activity. However, non-cached loads still incur energy consumption similar to integer code, indicating that power is not reduced during processor stalls during fetches from slow main memory.

To account for these differences when modeling task energy, we hypothesize that the P_{cpu} value when running a particular task will be the weighted average of these platform measurements, where the weights are based on the actual mix of instructions used by the task. Again, this is a great simplification since we categorize the instructions, disregard addressing modes, and do not really consider processor pipeline stalls due to memory fetches or branch mispredictions. However, this should be sufficient to produce more accurate estimates than with the simple bimodal model discussed earlier. Of course, we can still apply the DVS estimation techniques on top of the instruction mix model of P_{cpu}. We will evaluate how well this model works in Section 6.4.

6.3.5 Applying Models to New Platforms

So far, we have developed several models for platform power dissipation. In order to use the models to predict task energy consumption, we need to determine a few platform-specific parameters. Here, we outline the steps to measure these parameters.

First, a measure of power dissipation with the processor halted is needed. We can simply apply our task-energy measurement hardware to find this power constant, P_{sys}. If the system is not running anything else, one can use other methods of measuring energy as well, as the complexities of synchronizing measurements to a task are not relevant.

Next, to obtain P_{cpu} for the bimodal model, we construct a long sequence of simple integer operations, such as register to register additions, and run this in an infinite loop. Subtracting P_{sys} from the power measured when executing this loop will provide P_{cpu} for the bimodal model.

If the instruction mix extension is to be used, one needs to identify categories of common instructions and repeat this measurement for each, using a similarly-constructed loop. Our categorization of instructions into integer, floating point, load/store, and branch should suffice for most architectures, but one may need additional processor-dependent categories, such as for vector operations.

To account for DVS in the models, we do not need additional measurements. Instead, specifications of the available frequency and voltage settings suffice. To use the model as stated, the above measurements should be performed with the processor running at the highest frequency and voltage settings available on the platform.

These measurements need to be performed only once on the platform. Using these, an application developer can apply our models to predict task execution energy without further energy measurement. The application task needs to be profiled to determine the execution time and the mix of processor instructions used. Then, using the platform parameters, the developer can estimate P_{cpu} as the weighted average of the instruction-specific measures, and applying the bimodal model, determine the execution energy on the platform. For other platforms using the same instruction set (i.e., that do not need recompiling), simply plug-in the new platform parameters. If recompiling is necessary, then the profiling step needs to be repeated to find the appropriate weights for each instruction type.

	K6 laptop	Athlon laptop
P_{sys}	13.1 W	17.6 W
$P_{cpu-max}$:		
integer	15.3 W	15.6 W
floats	17.2 W	14.9 W
load/store	17.3 W	18.2 W
branch	15.5 W	15.8 W

Table 6.2: Platform measurements of model constants on two test machines.

6.4 Evaluation

To maximize the benefits of computation for a given energy budget using task-adaptation techniques, one needs to specify accurate measures of task energy consumption on the execution platform. The most accurate values for task energy are obtained through direct measurements at a time-slice granularity, as with the hardware we have proposed. Because it is not always practical to measure all applications on all possible hardware configurations, we have proposed some simple models to predict energy consumption. In this section, we evaluate the accuracy of energy models on real hardware, using real applications.

6.4.1 Platform Characteristics

We use two different laptops as the platforms to evaluate the task energy models. The first is the HP N3350, an AMD K-6 based laptop used earlier in the development of the models. The second is a Compaq Presario 1200Z that uses the mobile Athlon processor. These particular processors were chosen because they are capable of dynamic voltage scaling [2, 3], and can, therefore, be used to test our model of DVS effects on energy consumption.

To use our models with these platforms, we need to perform a few measurements to determine P_{sys} and P_{cpu} values, as outlined in Section 6.3.5. We make the energy measurements when running the processors at the maximum frequency and voltage on each laptop: 550 MHz at 2.0V for the K6 laptop, and 1 GHz at 1.4V for the Athlon laptop. The results of these measurements are summarized in Table 6.2, indicating power dissipation when the processor is halted, as well as the additional power dissipation when executing the four categories of instructions we identified earlier.

Instructions	GCC	GNUplot	Mpg123
integer	35.5%	33.9%	47.7%
floating point	0.0%	4.8%	1.9%
load/store	45.0%	44.6%	46.2%
branch	19.6%	16.7%	4.2%

Table 6.3: Instruction mix for test applications.

6.4.2 Application Characteristics

To best evaluate our models, we run real applications on the chosen hardware platforms. We picked three common desktop applications: the GCC compiler, the GNUplot graphing application, and the Mpg123 audio player. Although these are not necessarily the types of applications that are commonly executed on mobile platforms, we use these because they are readily available and provide different types of computationally-intensive workloads.

The GCC test involves compiling a C program, with optimizations enabled. Compilation is very CPU intensive, limited by processing speed, and not by file access. To eliminate completely any disk interference, we use a ramdisk to hold source, destination, and temporary files. The computational load due to compilation primarily involves string processing, i.e., integer operations, and is generally characterized as branch intensive.

For the second task, we use the GNUplot application's regression analysis feature to fit a quadratic polynomial to a set of test points. Unlike the GCC test case, we expect this task to use floating point operations in significant proportions. Here, too, we keep the data files in a ramdisk to avoid disk activity.

Finally, the third test task uses Mpg123, an open-source MPEG 1, Layer 3 audio player, to uncompress an audio file kept in a ramdisk. We pipe the output to the null device, so the program executes continuously, as fast as possible, rather than run intermittently to keep pace with audio output during normal playback.

In order to use our instruction mix power model to predict the energy consumed in executing these tasks, we need to determine the fraction of executed operations falling into each of the instruction categories described earlier. This cannot be done statically, as the ratios of executed operations can be greatly different from the ratios in the binary images, due to looping and OS service calls. To determine the mix of instructions dynamically, we execute these tasks using the SIMICS simulator [51, 89], and dump several traces of the actual operations executed. Parsing these multi-million-instructions-long traces, we determine the instruction ratios shown in Table 6.3. Note that the load/store entry includes integer operations for which the addressing mode involves a memory operand. With this information, we can now apply our models and compare the estimates to actual measurements of task energy.

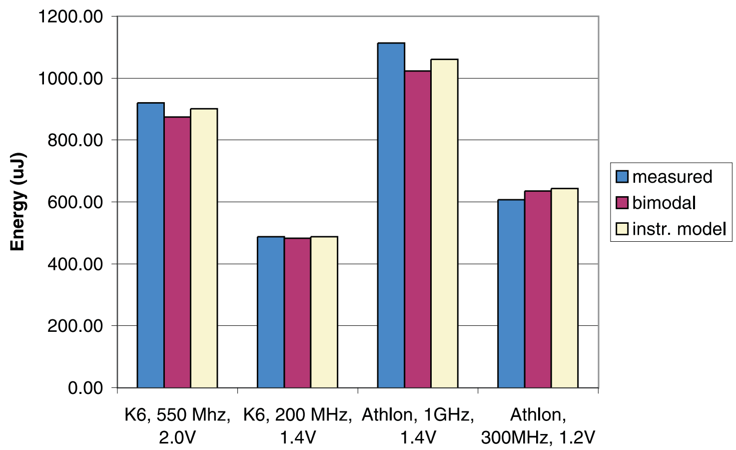
6.4.3 Experimental Results

We execute the three different test tasks on the two platforms described above, and use our charge flow measurement hardware to measure the energy consumed executing the tasks. For the GCC and Mpg123 test cases, the tasks execute for significantly longer than one scheduling time-slice and beyond the direct measurement capability of our hardware, so we take energy measurements over random intervals ranging from 10 to 50 ms during the execution of the tasks. For the GNUplot case, the execution of the polynomial regression fit function is fast enough that we simply measure the energy consumed over its entire execution.

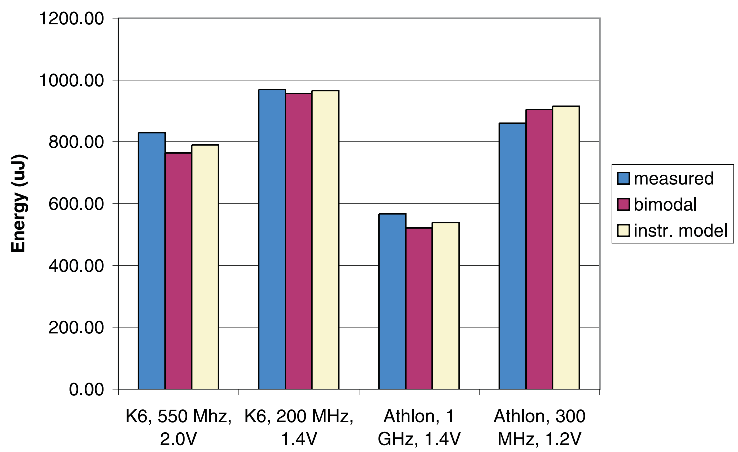
This measurement is performed on both laptops, in two configurations for each at the extremes of the available frequency and voltage settings. By taking and averaging over 50-100 measurements for each task for each hardware configuration, we obtain the measured energy plotted in Figure 6.8. In addition, using the time intervals of the measurements, the task instruction mix data, and the platform characteristics, we also compute and plot the energy consumption predicted by the bimodal and instruction mix power models, when used in conjunction with our model of DVS effects.

On initial inspection, these results show that the models can predict execution energy very closely. The instruction mix model is within 5% of the actual measurements for both the GCC and GNUplot tests, except for the Athlon platform at reduced voltage. But even here, it is only 5.9% and 6.4% off for the two tests, respectively. This may be explained due to the fact that the Athlon laptop seems to deviate slightly from our DVS energy model — when the voltage is reduced, the measured P_{sys} when the processor is halted reduces as well, considerably more so than the K6 laptop, while our model assumes this stays constant. As a result, although the models generally underestimate the energy needed, they overestimate for the Athlon laptop when CPU voltage is reduced. Although not as effective

GCC



GNUplot



Mpg123

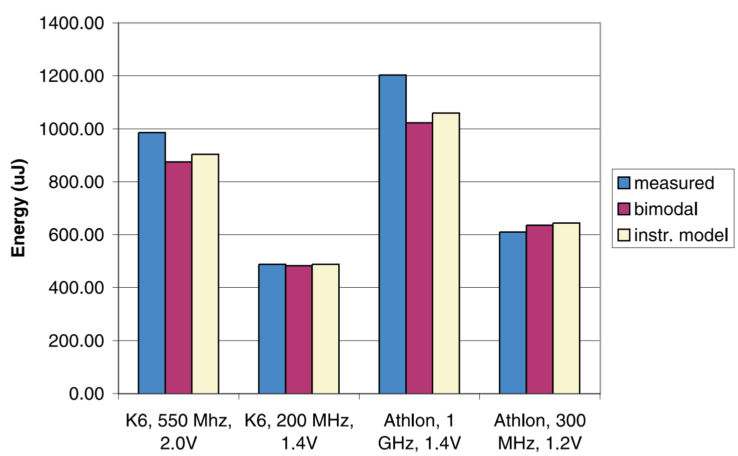


Figure 6.8: Measurements of task energy, compared to model estimates.

as the instruction mix model, the bimodal model is within approximately 8% for the GCC and GNUplot tests.

The Mpg123 test, however, presents more of a problem, and both models exhibit greater error in estimating energy. The instruction mix model is up to 12% off here. This may be due to the fact that this task uses highly optimized decoding algorithms that, though effective use of the instruction set, increase execution parallelism, instruction thoughput, and, therefore, power consumption. Our simplified models cannot take this increased parallelism into account, and underestimate the power consumption. This optimization may also explain why there are so few branch instructions executed in this task, at only 4% of the executed operations, when compared to the other two tasks that have a 4-5 times higher frequency of branches. Note that in the low-voltage Athlon case, the DVS model error offsets this underestimation, reducing the total estimation error.

Overall, the models seem to perform very well, predicting most of our test cases within 5% of the measured values, and even in the worst of these cases, within a 15% error.

6.5 Related Work

There have been numerous efforts dealing with energy consumption in mobile and handheld devices. Most are related to hardware energy management. Less work has been done in terms of OS support and management for energy reduction, although there have been some calls to action on energy management in software systems [18]. Recently, many researchers have focused on high level control of dynamic voltage scaling (DVS) to reduce energy, both for general purpose [20, 47, 68, 69] and real-time systems[24, 36, 56, 65], although the first DVS related work appeared some time ago [90].

The adaptation of task sets to available resources has been studied by various researchers, and Noble, *et. al* [58], have produced a general adaptation framework for mobile platforms. This has later been extended [21] to perform energy adaptation of workloads and meet runtime requirements on limited energy supplies. More recently, algorithms have been devised to adapt task sets based on the trade-off between task energy consumption and utility or rewards of task execution [72, 73], as in the EQoS framework presented in Chapter 5. Particularly for this last type of adaptation, it is necessary to know the energy requirements of all tasks in the system.

More directly related to this work, Lorch, *et. al* [48], were among the first to measure system energy consumption on a laptop, using built-in power monitoring devices. These techniques cannot provide the time granularity of measurements needed to determine task energy consumption. Flinn, *et. al* developed the PowerScope methodology, which allows one to statistically determine the energy profile of a task after repeated executions, but it does require some invasive modifications to the running system to synchronize measurements to task execution.

The closest measurement scheme to our charge flow meter is presented in [7, 8]. These works use a custom board and a set of switched capacitors to supply power to an on-board processor or microcntroller that is measured. Their system is tailored for high-frequency measurements of energy during each individual processor cycle, and not for the measurements over time intervals corresponding to task execution as we desire. This also requires expensive hardware to perform the high-speed data acquisition, requires the processor to be on-board rather than in the target platform, and suffers from the same problems as oscilloscope-based measurements — the time granularity is too fine, requiring huge amounts of data to be collected and integrated to do a single measurement of a task's energy consumption.

Based on these works, a model for predicting execution energy on RISC processors has been developed [40]. This requires intensive cycle-accurate measurements of the target processor and require very detailed information regarding the executed instructions, such as the fetch address and opcode bits, to predict the processor energy for relatively short sequences of instructions with a 2.5% error on average. As with the previous works, this model is at too fine a time granularity to be practical when applied to the millions of instructions executed in just a single execution time-slice. In contrast, our models of task energy are easily applied to arbitrarily long instruction sequences, and even with great simplification, can on average provide energy estimates with only a 5% error.

6.6 Conclusions

In this chapter, we have addressed how to determine task energy consumption, an important issue in any low-power system design, and critical to any systems that employ energy-aware task adaptation. We have proposed and implemented a low-cost, microcontroller-based energy-consumption measurement device that addresses the problems of measuring energy over the millisecond-range execution times of tasks and synchronizing measurements with task execution. This is accomplished without relying on invasive modifications of the target systems or expensive external hardware, and we propose methods of automating measurements to create a self-monitoring platform.

In addition to directly measuring energy consumption, we have developed simple models to predict task execution energy based on platform and task characteristics. We have shown these to be accurate, particularly our instruction mix model, which comes within 5% of actual measurements in most of the test applications. Overall, these estimates should be sufficient for use in task adaptation systems, which only require the average execution energy specifications of each task.

Although they work very well, the models are not perfect due to the simplifying assumptions we have made. In the future, we would like to investigate how the models can be improved, particularly in regard to predicting increases in energy consumption due to improved instruction-level parallelism when executing optimized code on superscalar processors. In addition, we currently model only the processor energy consumption, which is appropriate for computationally intensive tasks on platforms where CPU power dominates, but not effective for systems where non-CPU components, such as wireless transmitters or mechanical actuators, dominate power consumption. We plan to further develop models that can also incorporate I/O and communication hardware energy consumption.

CHAPTER 7

Conclusions

As computer technology progresses making smaller, lighter, and faster computing devices available at inexpensive prices, we are experiencing a proliferation of small, non-traditional computing platforms that are capable of high computational performance and sophisticated functions. The exponential improvements in processing speed and similar gains in large-scale integration allow handheld and embedded systems today to be as computationally powerful as workstation-class machines of just a few years ago. However, despite improving battery and semiconductor technologies, providing the energy needed for high-performance computing in a small package is a significant challenge. To alleviate the problem, we must consider methods and technologies to reduce energy consumption and improve the energy efficiency of systems.

This thesis has taken a software-centric approach to energy conservation in embedded and mobile systems, where the system may face both energy and timeliness of execution constraints for a real-time task set. The general approach to energy conservation has been divided into three major aspects: improve efficiency of OS services by targeting and optimizing them specifically for small embedded systems; develop and implement algorithms to best utilize and exploit hardware energy-conserving mechanisms while maintaining RT deadline guarantees; and adaptation of the real-time workload to maximize the returns on limited stored energy available in a system. The major focus of this thesis is on the latter two aspects, particularly in the context of reducing energy consumed by the processor.

7.1 Contributions

In this thesis, we have explored various methods of making an embedded, real-time operating system more energy-aware and efficient. The major contributions of this thesis are summarized as follows:

- After dividing the software centric approach to energy conservation into three aspects, this thesis first surveys various techniques through which system services may be modified to more closely match the needs of embedded real-time systems, making the general-case operation lower in processing overheads and energy consumed. New experiments extending one particular mechanism — protocol layer bypassing with a zero-copy architecture — to modern architectures are shown to save up to 65% of the processing energy costs of receiving a packet in multimedia streaming applications that need the connectivity of the Internet Protocol, but none of its fragmentation and error correction / detection services.

- Software-controlled power-down of the processor core and external components can significantly reduce energy consumption. However, due to potentially large power-state switching latencies, using such mechanisms in a real-time system can cause task deadline violations. *Sprint-and-halt* scheduling techniques are developed to exploit hardware power-down mechanisms and take these latencies into account to conserve energy, while maintaining real-time execution guarantees. Several algorithms of increasing complexity are described and evaluated, showing 40–70% reduction in energy over a broad range of system parameters, while maintaining real-time performance.

- Hardware techniques of dynamically lowering the operating voltage of the processor can greatly reduce processing energy costs, but will affect execution timing as the operating clock frequency must be reduced in tandem. This thesis develops novel *real-time dynamic voltage scaling* (RT-DVS) algorithms that can provide deadline guarantees while adjusting processor frequency and voltage to save energy. Several algorithms are developed, for both static and dynamic priority real-time scheduling, and are shown to conserve 20–40% of processing energy in addition to any savings from an ideal processor halt mechanism.

- A working implementation of these RT-DVS algorithms has been developed on top of the Linux 2.2 kernel, using AMD K6 and Athlon mobile Athlon processors. This is one of the first implementations of DVS published, and uses a readily available and modifiable OS and hardware platform, so is a good base for future DVS (real-time, or otherwise) experimentation. It is used in this thesis to verify and validate the simulation methodology and results, showing a comparable measured energy savings due to RT-DVS.

- The third aspect of software-centric energy conservation is explored in this thesis through the development of the Energy-Aware Quality-of-Service (EQoS) framework. This framework allows the system to provide differentiated service, in terms of energy and processing resources, to each real-time application in a system, in order to maximize the total value or utility gained from a limited supply of stored energy. It frames the adaptation of the working task set to the available energy into a tractable multiple-choice knapsack problem (MCKP) with straightforward optimal solutions, and very efficient heuristics. EQoS provides the maximum benefits on a limited energy budget, while meeting real-time constraints and runtime goals. The RT-DVS implementation has been extended to support EQoS.

- DVS can greatly increase the processing energy efficiency of a system that is under-loaded, so using it when adapting the workload on a system can significantly skew the intended results of adaptation. The concept of an idealized DVS response is introduced to model the effects that DVS techniques have on system runtime when workload is reduced. Using this relation, EQoS techniques can account for DVS effects on the task set as a whole, and can continue the independent treatment of tasks in a tractable MCKP problem.

- In order to use the EQoS framework for task adaptation, one must know the energy requirements of each task. This thesis also looks at support technology that addresses measuring and modeling task energy requirements. A small, low-cost device is developed that can provide direct energy measurements of individual tasks, without the shortcomings or expense of most current approaches. A self-monitoring architecture is described that allows a system to monitor and update energy profiles of tasks for online adaptation. Finally, simple models of task energy are proposed that can predict average energy consumption within 5% of actual requirements.

7.2 Future Directions

Together, the techniques developed in this thesis can be used to significantly improve the power dissipation and energy efficiency of a mobile, embedded system. There are some caveats in the integration of these that can provide opportunities for future research. In particular, the philosophies behind sprint-and-halt and RT-DVS scheduling are mutually opposed — one tries to get all work done as quickly as possible, while the other tries to spread it out maximally. One future direction is to devise a hybrid scheduler that automatically switches between these modes of scheduling based on the expected surplus processing capacity and energy savings possible through voltage scaling and powering-down of the system.

Furthermore, the work in this thesis is primarily concerned with the energy consumed through the microprocessor, and focuses on reducing computational overheads, or improving the energy-efficiency of computation. Future research can direct attention to components external to the processor. In particular, although it does not draw as high a peak power as the processor, the memory subsystem generally draws a continuous, low, but significant amount of power in small embedded systems. Mechanisms to dynamically reduce the power consumption of memory while maintaining performance (particularly real-time aspects) should be investigated [26].

One key influence on handheld device battery life is the energy costs associated with communications, particularly through wireless LANs or cellular networks. Although reducing energy consumption of wireless networking in well studied [35, 81], how such techniques can be incorporated into real-time scheduling, and sprint-and-halt mechanisms is an open problem. Furthermore, in applications such as streamed multimedia, there is often a tradeoff between the size of the data stream and the processing necessary to decode / uncompress it, so reducing communication bandwidth and energy costs may require increases in processing energy. Handling this tradeoff may be a future extension to the EQoS framework.

Currently, the EQoS framework requires that the tasks in the system be independent of each other with respect to the value gained from execution at particular service levels. A future avenue of research is to extend the system to permit dependencies, such as “task B has value only if task A and task C are executing.” Handle such expressive dependencies requires significant additional research. In particular, the optimization problem will no longer reduce to a multiple-choice knapsack problem, so further investigation into algorithms is necessary in order to avoid exponential search of the entire solution space for the workload adaptation.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Advanced Configuration and Power Interface. http://www.acpi.info/.

[2] Advanced Micro Devices Corporation. *Mobile AMD-K6-2+ Processor Data Sheet*, June 2000. Publication # 23446.

[3] Advanced Micro Devices Corporation. *Mobile AMD Athlon 4 Processor Model 6 CPGA Data Sheet*, Nov. 2001. Publication # 24319E.

[4] Atmel Corporation. http://www.atmel.com/products/avr/.

[5] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VSLI*, pages 299–316, June 2000.

[6] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In T. N. Mudge and B. D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 288–297, Los Alamitos, CA, USA, January 1995. IEEE Computer Society Press.

[7] N. Chang and K. Kim. Real-time per-cycle energy consumption measurement of digital systems. *IEE Electronics Letters*, 36(13):1169–1171, June 2000.

[8] N. Chang, K. Kim, and H. G. Lee. Cycle-accurate energy consumption measurement and analysis: Case study of arm7tdmi. *IEEE Transactions on VLSI Systems*, 10:146–154, April 2002.

[9] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, Department of Computer Science, University of York, 1997.

[10] V. Cherkassky and M. Malek. Graceful degradation of multiprocessor systems. In *International Conference on Parallel Processing*, pages 885–888, Pennsylvania, Pa, USA, August 1987. Pennsylvania State Univ. Press.

[11] H. K. J. Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[12] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Networks*, 7(4):36–43, July 1993.

[13] J. K. Dey, D. F. Towsley, C. M. Krishna, and M. Girkar. Efficient on-line processor scheduling for a class of iris real-time tasks. In *SIGMETRICS*, pages 217–228, 1993.

[14] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. Operating Systems Design and Implementation*, October 1996.

[15] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 189–202. ACM Press, December 1993.

[16] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints. In *ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, June 2002.

[17] K. Dudzinski and S. Walukiewicz. Exact methods for the knapsack problem and its genralizations. *European Journal of Operational Research*, 28:3–21, 1987.

[18] C. S. Ellis. The case for higher-level power management. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 162–167, Rio Rico, AZ, March 1999.

[19] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, August 1996.

[20] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, Rome, Italy, July 2001.

[21] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 48–63, Kiawah Island, SC, December 1999. ACM Press.

[22] J. Flinn and M. Satyanarayanan. PowerScope: a tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, New Orleans, LA, February 1999.

[23] K. Govil, E. Chan, and H. Wassermann. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95*, March 1995.

[24] F. Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Huntington Beach, CA, August 2001.

[25] H. Huang, P. Pillai, and K. G. Shin. Imroving wait-free algoritms for interprocess communication in embedded real-time systems. In *USENIX Annual Technical Conference*, June 2002.

[26] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proc. of USENIX Annual Technical Conference (USENIX'03)*, San Antonio, TX, June 2003.

[27] C. Hwang and A. C.-H. Wu. A predictive system shut-down method for energy saving of event-driven computation. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 28–32, 1997.

[28] Intel Corporation. http://developer.intel.com/design/intelxscal/.

[29] Intel Corporation. *Mobile Intel Pentium III Processor in BGA2 and MicroPGA2 Packages*, 2000. Order Number 245483-003.

[30] Intel, Inc. Pentium processor. ftp://download.intel.com/design/pentium/datashts/24199710.pdf, June 1997.

[31] Intel, Inc. Pentium II processor at 350 MHz, 400 MHz, and 450 MHz. ftp://download.intel.com/design/PentiumII/datashts/24365703.pdf, August 1998.

[32] M. Kim and B. D. Noble. Mobile network estimation. In *Proc. of 7th ACM Conference on Mobile Computing and Networking (MOBICOM'01)*, Rome, Italy, July 2001.

[33] W. Kim, D. Shin, H.-S. Yun, J. Kim, and S. L. Min. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, 2002.

[34] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: a solution to a real-time synchronization problem. In *Proc. Real-Time Systems Symposium*, pages 131–137, 1993.

[35] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98)*, pages 157–168, New York, October 1998. ACM Press.

[36] C. M. Krishna and Y.-H. Lee. Voltage-clock-scaling techniques for low power in hard real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 156–165, Washington, D.C., May 2000.

[37] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[38] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[39] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete qos options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE, June 1999.

[40] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded risc processors. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, pages 1–10, Snowbird, UT, June 2001.

[41] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[42] J. Lehoczky and S. Thuel. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1994.

[43] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. of the 8th IEEE Real-Time Systems Symposium*, pages 261–270, Los Alamitos, CA, December 1987.

[44] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

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

[46] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. In *Proceedings of the IEEE*, pages 83–93, January 1994.

[47] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 50–61, Cambridge, MA, June 2001.

[48] J. R. Lorch and A. J. Smith. Apple Macintosh's energy consumption. *IEEE Micro*, 18(6):54–63, Nov. 1998.

[49] J. R. Lorch and A. J. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.

[50] Y.-H. Lu, L. Benini, and G. D. Micheli. Low-power task scheduling for multiple devices. In *International Workshop on Hardware/Software Codesign*, pages 39–43, May 2000.

[51] P. S. Magnusson et al. Simics/sun4m: A virtual workstation. In *Proc. of Usenix Annual Technical Conference (USENIX'98)*, New Orleans, LA, June 1998.

[52] S. Martello and P. Toth. *Knapsack Problems*. John Wiley and Sons, Ltd., 1990.

[53] Maxim Integrated Products. *Precision, High-Side Current-Sense Amplifiers*, December 1996. Datasheet 19-0335, Rev. 2.

[54] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Symposium on Operating System Principles*, pages 39–51. ACM Press, Nov. 1987.

[55] S.-W. Moon, P. Pillai, and K. G. Shin. STREAMER: hardware support for smoothed transmission of stored video over atm. In *Parallel Computer Routing and Communication Workshop*, Atlanta, GA, June 1997.

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

[57] R. M. Nauss. The 0-1 knapsack problem with multiple choice constraints. *European Journal of Operational Research*, 2:125–131, 1978.

[58] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France, October 1997.

[59] T. Pering and R. Brodersen. Energy efficient voltage scheduling for real-time operating systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS'98, Work in Progress Session*, Denver, CO, June 1998.

[60] T. Pering and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'98*, pages 76–81, Monterey, CA, August 1998.

[61] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 76–81, New York, August 10–12 1998. ACM Press.

[62] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'00*, Rapallo, Italy, July 2000.

[63] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[64] P. Pillai. http://kabru.eecs.umich.edu/rtos/eqos.tar.gz.

[65] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102, Banff, Alberta, CA, October 2001.

[66] P. Pillai and K. G. Shin. Sprint-and-halt scheduling for energy reduction in real-time systems with software power-down. Technical Report CSE-TR-482-03, Computer Science and Engineering, University of Michigan, 2003.

[67] D. Pisinger. The multiple-choice knapsack problem. *European Journal of Operational Research*, 83:394–410, 1995.

[68] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, Rome, Italy, July 2001.

[69] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Huntington Beach, CA, August 2001.

[70] J. Pouwelse, K. Langendoen, and H. Sips. Application-directed voltage scaling. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2002.

[71] P. Ramanathan. Graceful degradation in real-time control applications using (m, k)-firm guarantee. In *IEEE FTCS 27*, pages 132–143, 1997.

[72] C. Rusu, R. Melhem, and D. Mosse. Maximizing rewards for real-time applications with energy constraints. to appear in *ACM Transaction on Embedded Computing Systems*.

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

[74] SBS Implementers Forum. *Smart Battery Data Specification, Revision 1.1*, December 1998. http://www.sbs-forum.org.

[75] D. B. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *IEEE RTSS 96*, pages 13–21, 1996.

[76] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers*, 39(3):1175–1198, 1990.

[77] W.-K. Shih. Scheduling in real-time systems to ensure graceful degradation: The imprecise-computation and the deferred-deadline approaches. Technical Report 1765, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1992.

[78] K. G. Shin and C. L. Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *11th Euromicro Conf. on Real-Time Systems*, 1999.

[79] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[80] J. Stankovic et al. *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 1998.

[81] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications, vol.E80-B, no.8, p. 1125-31*, E80-B(8):1125–31, 1997.

[82] V. Swaminathan and K. Chakrabarty. Real-time task scheduling for energy-aware embedded systems. In *Proceedings of the IEEE Real-Time Systems Symp. (Work-in-Progress Session)*, Orlando, FL, Nov. 2000.

[83] V. Swaminathan, K. Chakrabarty, and S. S. Iyengar. Dynamic i/o power management for hard real-time systems. In *Proc. Intl. Symposium on Hardware/Software Co-Design (CODES)*, pages 237–242, 2001.

[84] H. Takada and K. Sakamura. Experimental implementations of priority inheritance semaphore on ITRON-specification kernel. In *11th TRON Project International Symposium*, pages 106–113, 1994.

[85] The LAME Project. http://www.mp3dev.org/mp3/.

[86] H. Tokuda and T. Nakajima. Evaluation of real-time synchronization in Real-Time Mach. In *Second Mach Symposium*, pages 213–221. Usenix, 1991.

[87] Transmeta Corporation. http://www.transmeta.com/.

[88] S.-Y. Tzou and D. P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software – Practice and Experience*, 21(3):251–267, March 1991.

[89] Virtutech. http://www.simics.com/.

[90] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, Monterey, CA, Nov. 1994.

[91] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1995.

[92] E. Zemel. The linear multiple choice knapsack problem. *Operations Research*, 28(6):1412–1423, 1980.

[93] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):686–700, 2003.

[94] K. M. Zuberi. *Real-Time Operating System Services for Networked Embedded Systems*. PhD thesis, University of Michigan, 1998.

[95] K. M. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: A small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 277–291, Kiawah Island, SC, December 1999. ACM Press.

[96] K. M. Zuberi and K. G. Shin. An efficient end-host protocol processing architecture for real-time audio and video traffic. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 111–114, July 1998.