ENERGY EFFICIENT SYSTEM DESIGN AND UTILIZATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Tajana Šimunić
February 2001

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Teresa Meng

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Steven Boyd

Approved for the University Committee on Graduate Studies:

Abstract

Energy consumption of electronic devices has become a serious concern in recent years. Energy efficiency is necessary to lengthen the battery lifetime in portable systems, as well as to reduce the operational costs (e.g. cost of electricity) and the environmental impact (e.g. cooling fan noise) of stationary systems. Optimization in design and utilization of both hardware and software is needed in order to achieve more energy efficient systems.

In this thesis I first discuss power management algorithms that enable optimal utilization of hardware at run time. Next, I discuss the new dynamic voltage scaling algorithm that complements power management by scaling processor frequency and voltage depending on the needs of the system. Finally, I develop a modular approach for design and simulation of hardware and software energy consumption at the system level.

Dynamic power management (DPM) algorithms aim to reduce the energy consumption at the system level by selectively placing components into low-power states. Two power management algorithms will be presented that have been derived from the stochastic models of several realistic examples. Both algorithms are event-driven and give optimal results verified by measurements. The first approach is based on renewal theory. This model assumes that the decision to transition to low power state can be made in only one state. Another method developed is based on the Time-Indexed Semi-Markov Decision Process model (TISMDP). TISMDP model assumes that a decision to transition into a lower-power state can be made upon each event occurrence from any number of states. On the other hand, it is also more complex than the approach based on renewal theory. It is important to note that the results obtained by renewal model are guaranteed to match results obtained by TISMDP model, as both approaches give globally optimal solutions. I implemented both power management algorithms on two different classes of devices: two different hard

disks and client-server WLAN systems such as the SmartBadge or a laptop. The measurement results show power savings ranging from a factor of 1.7 up to 5.0 with performance basically unaffected.

Dynamic voltage scaling (DVS) algorithms reduce energy consumption by changing processor speed and voltage at run-time depending on the needs of the applications running. In this work I extended the DPM model discussed above with a DVS algorithm, thus enabling larger power savings. I tested my approach on MPEG video and MP3 audio algorithms running on the SmartBadge portable device [51]. The results show savings of a factor of three in energy consumption for combined DVS and DPM approaches.

Lastly, I present a modular approach for enhancing instruction level simulators with cycle-accurate simulation of energy dissipation in systems. This methodology has tightly coupled component models thus making the simulation more accurate. Performance and energy computed by the simulator are within 5% tolerance of hardware measurements on the SmartBadge portable device. The simulation methodology can be used for hardware design exploration aimed at enhancing the SmartBadge with real-time MPEG video feature. In addition, a profiler that relates energy consumption to the source code has been developed. The MP3 audio decoder software has been redesigned using the profiler and the software design methodology to run in real time on the SmartBadge with low energy consumption. Performance increase of 92% and energy consumption decrease of 77% over the original executable specification have been achieved.

Dedication

To my husband, Dustin, and my families, with love.

Acknowledgments

This thesis would not have been possible without my advisor, Professor Giovanni De Micheli, who guided my efforts and encouraged me throughout the years spent at Stanford. Many thanks to Professor Peter Glynn, for hours we spent discussing the stochastic modeling concepts. I am very grateful to the members of my reading committee, Professor Teresa Meng and Professor Stephen Boyd, for the time and the patience spent in reading these pages.

Special thanks to Professor Luca Benini at DEIS, Universitá di Bologna for many discussions about my research, and for the hundreds of prompt email responses. I would like to thank all current and past members of the CAD group whose help and support have been invaluable. I am grateful to Yung Lu for his work on the experimental setup. In addition, I would like to thank Haris Vikalo, for both his input into my research work, and for his support as my good friend.

I would also like to thank Mark Smith and the members of his group at Hewlett-Packard Laboratory for their help with my research work: Malena Mesarina, for her help with the SmartBadge, Mat Hans, for his input on MP3 algorithms, John Apostolopoulos and Susie Wee for lending me a hand with MPEG video algorithms.

This work was sponsored in part by the Hewlett-Packard Laboratory, in part by NSF under grant CCR-9901190, and in part by the Marco Gigascale Research Center grant. Thank you for your economic support that made this work possible.

I am deeply in debt to my husband and my families for their love and support. Finally, many thanks to my friends near and far, for hanging in there with me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A system is a group of devices or objects whose interaction serves a common purpose. Many measures are used to evaluate how closely the design of electronic systems and subsystems meets the desired functionality and constraints. Some of the measures include performance, energy and power consumption, reliability, design and manufacturing cost. Power consumption is proportional to the frequency of execution and the square of the operating voltage, while energy consumption also depends on the total execution time.

Energy consumption has become one of the primary concerns in electronic design due to the recent popularity of portable devices and cost concerns related to desktops and servers. The battery capacity has improved very slowly (a factor of 2 to 4 over the last 30 years), while the computational demands have drastically increased over the same time frame. Heat extraction is a large issue for both portable and non-portable electronic systems. Finally, in recent time the operating costs for large electronic systems, such as data warehouses, have become a concern.

At the system level, there are three main sources of energy dissipation: (i) processing units; (ii) storage elements; (iii) interconnects and communication units. Energy efficient system level design must minimize the energy consumption in all three types of components, while carefully balancing the effects of their interaction. For example, optimizing the micro-architecture of a computing element can affect the energy consumption of memory and memory-processor busses. The software implementation also strongly affects the system energy consumption. For example, software compilation affects the instructions

used and thus the energy consumed by computing elements; software storage and data access in memory affect the energy balance between processing and storage units; and the data representation affects power dissipation of the communication resources.

Electronic systems are collections of components which may be heterogeneous in nature. For example, a laptop has a digital VLSI component, an analog component (wireless card), a mechanical part (hard disk drive), and an optical component (display). Peak performance in electronic design is required only during some time intervals. As a result, the system components do not always need to be delivering peak performance. The ability to enable and disable components, as well as of tuning their performance to the workload (e.g., user's requests), is important in achieving energy efficient utilization.

In this work I will present new approaches for lowering energy consumption in both system design and utilization. This work has been motivated by my experience in achieving energy efficient design and utilization of the SmartBadge and its components, and by optimizing the utilization of the hard disks and the WLAN card used in a laptop and a desktop computer.

1.1 SmartBadge

The SmartBadge, shown in Figure 1.1, is an embedded system consisting of Sharp's display, *wireless local area network* (WLAN) card, StrongARM-1100 processor, Micron's SDRAM memory, FLASH memory, sensors, and modem/audio analog front-end on a *printed circuit board* (PCB) powered by the batteries through a DC-DC converter. The SmartBadge component diagram is shown in Figure 1.2. The initial goal in designing the SmartBadge was to allow a computer or a human user to provide location and environmental information to a location server through a heterogeneous network. The SmartBadge could be used as a corporate identity card, attached (or built in) to devices such as *personal digital assistants* (PDAs) and mobile telephones, or incorporated in computing systems. The SmartBadge operates as a part of a client-server system. Thus it initiates and terminates each communication session. As the SmarteBadge is battery operated, minimizing energy consumption via both careful design and utilization is critical. I used the prototype implementation of the SmartBadge on a PCB to evaluate the results of my work.

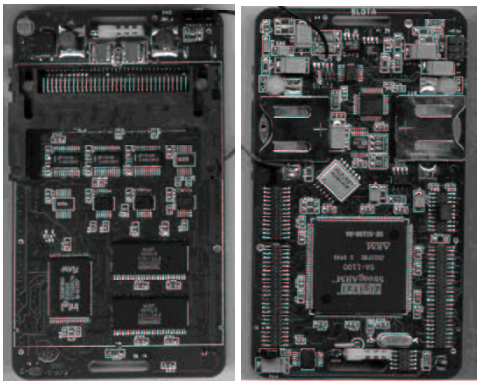

Figure 1.1: SmartBadge

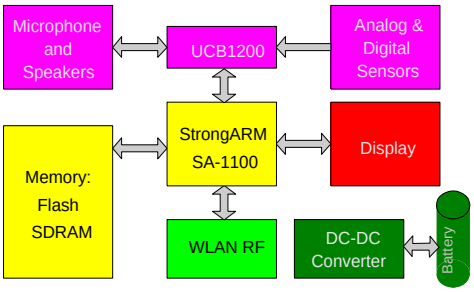

Figure 1.2: SmartBadge Components

The design goal is to enhance the prototype implementation of the SmartBadge so that it can support real-time MPEG video and audio decode. Since the original hardware does not meet either the performance or the energy consumption constraints when running the MPEG decode algorithm, both the hardware and the software architectures needed to be redesigned, while keeping the energy consumption under tight control. To address this, new hardware and software design methodologies have been developed. Once the design meets the performance constraints, it is important to consider the utilization of components. When full performance is not needed from the SmartBadge, the processor frequency and voltage can be scaled down using the dynamic voltage scaling algorithm presented in this thesis. Finally, as the SmartBadge is not constantly used, the whole system, or some of its components can be transitioned into low-power states using dynamic power management policy that is discussed in Chapter 2.

1.2 Sony Vaio Laptop

The Sony Vaio Laptop, shown in Figure 1.3 has been used daily in our lab. Its battery life, when no communication devices are used and the screen is dimmed, is about 2.5 hours. When a communication device is added to the system, such as WLAN card [50], the battery lifetime drops down by a factor of two. Most of energy consumption is taken by the display, hard disk and the WLAN card. When the laptop is being used, the display needs to be turned on. On the other hand, the hard disk and the WLAN card are not consistently accessed, and thus can be transitioned by the power management policy into low-power state when they are idle.

Figure 1.3: Sony Vaio Laptop (model PCG-F150)

The wireless card has multiple power states: two active states, *transmitting*, *receiving*, and two inactive states, *doze* and *off*. Transmission power is 1.65W, receiving 1.4W, the power consumption in the doze state is 0.045W [50] and in the off state it is 0W. Once both receiving and transmission are done, the card automatically enters the doze state. Unfortunately, savings of only 5-10% in power have been measured with this approach, due to the overhead of having to be awake every 100ms to find out if any communication needs to take place. In client-server systems, such as the laptop I used, it is clear when communication is finished on the client side. Thus, the power manager can turn the card off once the communication is finished, and turn in back on when the client wishes to resume communication. As that transition takes on average 60 ms, the overhead of turning the card off is not noticeable to the user.

In contrast, the performance overhead of transitioning the hard disk between active and

sleep states is considerable, about 2.2 seconds for the whole transition. In addition, it takes considerable power to spin up the disk - about 2.5 Watts. The average power consumption in the active state is $0.95W$, while in the sleep state, the disk consumes only $0.13W$. Thus, it is advantageous to place the disk into the sleep state only when the idle period between successive accesses is long enough to justify the performance and energy overhead incurred during the transitions. The dynamic power management algorithm I developed is able to give policies that give globally optimal decisions on when the transition to sleep state should occur. The new power management algorithm shows significant savings over the standard timeout algorithm typically implemented in systems.

Even though design is completed before optimizing the system utilization, the remainder of this chapter is organized as follows. Section 1.3 discusses the different approaches for energy efficient system utilization. Section 1.4 describes issues that are of concern when designing systems for lower energy consumption. Section 1.5 is a summary of the contributions of this thesis.

1.3 Energy Efficient System Utilization

While system design is concerned with selection and organization of system components, the system utilization addresses the question of how those components should be used. Electronic systems often consist of one or more microprocessors and a set of devices with multiple low-power states. Many microprocessors support dynamic clock frequency adjustment, and some newer devices also support dynamic supply voltage setting [24]. Thus, at the system level it is possible to reduce energy by transitioning components into low-power states (dynamic power management) and by changing the frequency and voltage level of the microprocessor (dynamic voltage scaling).

1.3.1 Dynamic Power Management

The fundamental premise for the applicability of power management schemes is that systems, or system components, experience non-uniform workloads during normal operation time. Non-uniform workloads are common in communication networks and in almost any

interactive system.

System-level *dynamic power management* [6] decreases the energy consumption by selectively placing idle components into lower power states. System resources can be modeled using state-based abstraction where each state trades off performance for power [35]. For example, a system may have an active state, an idle state, and a sleep state that has lower power consumption, but also takes some time to transition to the active state. The transitions between states are controlled by commands issued by a *power manager* (PM) that observes the workload of the system and decides when and how to force power state transitions. The power manager makes state transition decisions according to the *power management policy*. The choice of the policy that minimizes power under performance constraints (or maximizes performance under power constraint) is a constrained optimization problem.

In the recent past, several researchers have realized the importance of power management for large classes of applications. Chip-level power management features have been implemented in mainstream commercial microprocessors [23, 17, 22, 24]. Techniques for the automatic synthesis of chip-level power management logic are surveyed in [6].

The most common power management policy at the system level is a *timeout policy* implemented in most operating systems. The drawback of this policy is that it wastes power while waiting for the timeout to expire [43, 66].

Predictive policies for hard disks [15, 18, 26, 31, 49] and for interactive terminals [8, 34, 75] force the transition to a low power state as soon as a component becomes idle if the predictor estimates that the idle period will last long enough. An incorrect estimate can cause both performance and energy penalties. The distribution of idle and busy periods for an interactive terminal is represented as a time series in [75], and approximated with a least-squares regression model. The regression model is used for predicting the duration of future idle periods. A simplified power management policy predicts the duration of an idle period based on the duration of the last activity period. The authors of [75] claim that the simple policy performs almost as well as the complex regression model, and it is much easier to implement. In [34], an improvement over the prediction algorithm of [75] is presented, where idleness prediction is based on a weighted sum of the duration of past idle periods, with geometrically decaying weights. The policy is augmented by a technique

that reduces the likelihood of multiple mispredictions. All these policies are formulated heuristically, then tested with simulations or measurements to assess their effectiveness.

In contrast, approaches based on stochastic models can guarantee optimal results. Stochastic models use distributions to describe the times between arrivals of user requests (*interarrival times*), the length of time it takes for a device to service a user's request, and the time it takes for the device to transition between its power states. The system model for stochastic optimization can be described either with just memoryless distributions (exponential or geometric) [7, 14, 64, 65] or with general distributions [71, 72, 73, 74]. Power management policies can also be classified into two categories by the manner in which decisions are made: discrete time (or clock based) [7, 14] and event driven [64, 65, 71, 72, 73, 74]. In addition, policies can be stationary (the same policy applies at any point in time) or non-stationary (the policy changes over time). All stochastic approaches except for the discrete adaptive approach presented in [14] are stationary.

The optimality of stochastic approaches depends on the accuracy of the system model and the algorithm used to compute the solution. In both the discrete and the event-driven approaches optimality of the algorithm can be guaranteed since the underlying theoretical model is based on Markov chains. Approaches based on the discrete time setting require policy evaluation even when in low-power state [7, 14], thus wasting energy. On the other hand, event-driven models based on the exponential distribution [64, 65] show little or no power savings when implemented in real systems since the exponential model does not describe well the request interarrival times [71, 72, 73, 74]. In this thesis, I present two new approaches that combine the advantages of discrete and continuous models. The new DPM algorithms are guaranteed to be globally optimal, while allowing event-driven policy evaluation and providing a more flexible and accurate model for the user and the device.

1.3.2 Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) algorithms reduce energy consumption by changing processor speed and voltage at run-time depending on the needs of the applications running. If only processor frequency is scaled, the total energy savings would be small as power is inversely proportional to cycle time and energy is proportional to the execution time and

power. Early DVS algorithms set processor speed based on the processor utilization of fixed intervals and did not consider the individual requirements of the tasks running. There has been a number of voltage scaling techniques proposed for real-time systems. The approaches presented in [32, 33, 36, 86] assume that all tasks run at their worst case execution time (WCET). The workload variation slack times are exploited on task-by-task basis in [68], and are fully utilized in [45]. Work presented in [62] introduces a voltage scheduler that determines the operating voltage by analyzing application requirements. The scheduling is done at task level, by setting processor frequency to the minimum value needed to complete all tasks. For applications with high frame-to-frame variance, such as MPEG video, schedule smoothing is done by scheduling tasks to complete twice the amount of work in twice the allocated time.

In all DVS approaches presented in the past, scheduling was done at the task level, assuming multiple threads. The prediction of task execution times was done either using worst case execution times, or heuristics. Such approaches neglect that DVS can be done within a task or for single-application devices. For, instance, in MPEG decoding, the variance in execution time on frame basis can be very large: a factor of three in the number of cycles [5], or a range between 1 and 2000 IDCTs per frame [13] for MPEG video.

1.4 Energy Efficient System Design

Energy efficient system design requires the reduction of energy consumption in all portions of a system. System level design of hardware is concerned with selection and organization of the components. Software design is concerned with definition and selection of operating system, application software and compilers. The interaction between software and hardware components can greatly affect the energy consumption at the system level. Thus it is of critical importance to have a fast and easy way to evaluate energy consumption of the whole system during the design stages of software and hardware.

1.4.1 Energy Efficient Hardware Design

When designing an electronic system a designer explores a limited number of architectural alternatives and tests their functionality, energy consumption and performance. Traditionally, the designers aimed to meet high performance targets with little regard to energy consumption. Thus the whole design methodology aimed to deliver designs capable of delivering peak performance continually. Optimizing design for both performance and energy consumption has become a priority in recent times. As a result, the design methodology needs to be changed to include energy consumption criteria.

The most accurate way to evaluate the design is to build a prototype first, but this approach does not accurately model performance or energy consumption, is slow and very expensive. Alternatively, performance can be evaluated using instruction-set simulators (e.g., [1]), but there is limited or no support for energy consumption evaluation. Commercial tools target mainly functional verification and performance estimation [11, 16, 54, 76], but provide no support for energy-related cost metrics.

Processor energy consumption is generally estimated by *instruction-level power analysis*, first proposed by Tiwari et al. [78, 79]. This technique estimates the energy consumed by a program by summing the energy consumed by the execution of each instruction. Instruction-by-instruction energy costs are pre-characterized once for all for each target processors. The instruction-level power model can be augmented by considering the effect of first-level caches and inter-instruction effects. An approach proposed recently in [25] attempts to evaluate the effects of different cache and bus configurations using linear equations to relate the main cache characteristics to system performance and energy consumption. This approach does not account for highly non-linear behavior in cache accesses for different cache configurations that are both data and architecture dependent.

A few research prototype tools that estimate the energy consumption of processor core, caches and main memory in SOC design have been proposed [46, 42]. Memory energy consumption is estimated using cost-per-access models. Processor execution traces are used to drive memory models, thereby neglecting the non-negligible impact of a non-ideal memory system on program execution. The final system energy is obtained by summing over the contribution of each component. The main limitation of the approaches presented

in [42, 46] is that the interaction between memory system (or I/O peripherals) and processor is not modeled.

A more recent approach presented in [44] combines multiple power estimators into one simulation engine thus enabling detailed simulation of some components, while using high-level models for others. This approach is able to account for interaction between memory, cache and processor at run time, but at the cost of potentially long run-times. Longer run-times are caused by different abstraction levels of various simulators and by the overhead in communication between different components. The techniques that enable significant simulation speedup are presented, but at the cost of the loss of detail in software design and in the input data trace.

Cycle-accurate register-transfer level energy estimation is presented in [83]. This tool integrates RT level processor simulator with DineroIII cache simulator and memory model. It is shown to be within 15% of HSPICE simulations. This approach is not practical for component-based designs, as it requires knowledge of the internal design of system components.

An alternative approach for energy estimation using measurements as a basis for estimation is presented in PowerScope tool [21]. PowerScope requires two computers to collect the measurement statistics, some changes to the operating system source code and a digital multimeter. Although this system enables accurate code profiling of an existing system, it would be very difficult to use it for both hardware and software architecture exploration, as in the early design stages neither hardware nor operating systems or software are available for measurements.

Finally, most previous approaches do not focus on battery life optimization, the ultimate goal of energy optimization for portable systems. In fact, when the battery subsystem is not considered in energy estimation significant errors can result [52]. Some analytical estimates of the tradeoff between battery capacity and delay in digital CMOS systems are presented in [60]. Battery capacity is strongly dependent on the discharge current as can be seen from any battery data sheet [87]. Hence, it is important to accurately model discharge current as a function of time in an embedded system.

1.4.2 Energy Efficient Software Design

In the past, only performance and functionality of software were of concern. In recent years, the design trade-off of performance versus energy consumption has received large attention. Typically, for compatibility reasons, the degrees of freedom for modifying software are larger than those for hardware.

Since software does not have a physical realization, it is important to analyze the software impact on the hardware energy consumption. In addition, there is a need to evaluate which of different choices for the software implementation, such as system-level software or application-level software and their compilation into machine code, are most appropriate. Finally, it is crucial to be able to optimize software for both performance and energy. This can be done by changing the source code directly, and with the use of compilers.

Ideally, the goal is to develop energy-aware operating systems and applications, that can signal to hardware when they are needed and thus enable the most optimal performance and energy trade-off. Currently, the commercial software development tools only support performance profiling of software. Since the effect software has on the energy consumption is of critical importance, one of primary requirements for system design methodology is to effectively support code energy consumption optimization.

Several techniques for code optimization have been presented in the past. Tiwari et al. [78, 79] uses instruction-level energy models to develop compiler-driven energy optimizations such as instruction reordering, reduction of memory operands, operand swapping in the Booth multiplier, efficient usage of memory banks, and series of processor specific optimizations. In addition, several other optimizations have been suggested, such as energy efficient register labeling during the compile phase [53], procedure inlining and loop unrolling [46] as well as instruction scheduling [80]. Work presented in [41] applies a set of compiler optimizations concurrently and evaluates the resulting energy consumption via simulation. In [12] memory hierarchy is designed to better match software needs.

All the techniques discussed above focus on automated instruction-level optimizations driven by the compiler. Unfortunately, currently available commercial compilers have limited capabilities. The improvements gained when using standard compiler optimizations are marginal compared to writing energy efficient source code [70]. The largest energy

savings were observed at the inter-procedural level that compilers have not been able to exploit.

1.5 Thesis contribution

This thesis' contributions are in the areas of energy efficient system design and utilization. The system utilization is addressed with a combination of power management and voltage scaling algorithms. I present two new algorithms that give globally optimal power management policies for realistic component models. The policies have been implemented in real systems and have large measured power savings. The dynamic voltage scaling algorithm complements dynamic power management algorithms by changing processor frequency and voltage as the demands for processing power change.

Energy efficient system design methodology consists of hardware and software design methodologies. Energy conscious hardware design is done using a cycle-accurate energy consumption simulator presented in this thesis. The simulator enables fast and accurate evaluation of both hardware and software energy and performance. The simulator is complemented with the energy profiler that gives accurate estimates of energy consumption for each software function and by each hardware component. The software design methodology consists of two parts: the general approach that can be implemented in any system, and processor-specific examples that can be adopted to most systems.

More detailed overview of contributions of this thesis follows. First I start with the description of dynamic power management algorithms. Next, I discuss the dynamic voltage scaling algorithm. Finally, I give a quick overview of energy efficient system design techniques.

1.5.1 Dynamic Power Management

In this work I introduce two new models for power management at the system level that enable modeling system transitions with general distributions, but are still event driven and guarantee optimal results. The models assume that both devices and workloads for devices can be modeled using stationary stochastic distributions and the states describing

trade-offs in cost and performance. Thus, the applicability of these models is very general. For systems that do not exhibit stationarity, adaptive methods, such as the ones presented in [14] can be used.

In order to verify the models, I implemented the power management algorithms on general purpose systems, such as laptops and desktops, and on a portable system such as the SmartBadge [51]. On general purpose systems two main classes of devices were controlled with power management policies: storage devices, such as hard disks, and communication devices, such as the WLAN card. For each of these devices, I collected a sets of traces that model well typical user behavior. I found the interarrival times between user requests are best modeled with a non-exponential distribution (a Pareto distribution shows the best fit, although my model applies to any distribution or direct data). These results are consistent with the observations on network traffic interarrival times presented in [59]. In addition, I measured the distributions of transition times between active, idle and low power states for each of the systems and found non-exponential transition times into or out of a low power state. Traditional Markov chain models presented in previous work do not apply to these devices since user request arrivals and the transition times of a device are best modeled with the non-exponential distributions that can occur at the same time. As a result, I formulated the policy optimization problem using two different stochastic approaches. My models are a generalization of discrete-time Markov decision process model (DTMDP) presented in [7] and of continuous-time Markov decision process model presented in [64, 65]. Both DTMDP and CTMDP approaches use memoryless distributions to model all of the system behavior (geometric for DTMP and exponential for CTMDP).

The first approach I present is based on renewal theory [67, 77]. It is more concise, but also is limited to systems that have only one decision state. The second approach is based on Time-Indexed Semi-Markov Decision Process model (TISMDP). This model is more general but also more complex. In both cases the policy optimization problem can be solved *exactly* and in polynomial time by solving a linear program. Clearly, since both approaches guarantee optimal solutions, they will give the same solution to a given optimization problem. Note that both approaches can handle general user request interarrival distributions, even though in the particular examples presented in this work we use Pareto

distribution since it showed a good fit to the data collected experimentally. The policy decisions are made only upon request arrival or upon finishing serving a request, instead of at every time increment as in discrete-time model. Since policy decisions are made in event-driven manner, more power is saved by not forcing policy re-evaluations as in discrete-time models.

I implemented both policies in real systems and compared the resulting power consumption. The measurement results show that the reduction in power can be as large as 2.4 times with a small performance penalty when power managing the laptop hard disk and 1.7 times for the desktop hard disk. My algorithms perform better than any other power management algorithms tested in [48]. The measurements of optimal policy implemented on a laptop for the WLAN card show that the reduction in power can be as large as a factor of 5 with a small performance penalty. Finally, power management results on the SmartBadge show savings of as much as 70% in power consumption.

1.5.2 Dynamic Voltage Scaling

In this work I extend the DPM model with a DVS algorithm, thus enabling larger power savings. The DVS algorithm assumes that the exponential distributions can be used to model the workload while the system is in the active state. In addition, it represents the active state as a series of states characterized by varying degrees of performance and energy consumption.

The algorithm is implemented for the SmartBadge portable device [51]. The SmartBadge processor can operate over a range of frequencies. For each frequency, there is a minimum allowed voltage of operation. If the processor is run at the minimum frequency and voltage required to sustain the performance level required by the application, it is possible to save power even when the system is active, in addition to the savings that can be obtained by DPM during idle periods. This principle is exploited by the recently announced Transmeta's Crusoe processor [24].

A first contribution is to develop and verify a stochastic model for prediction of execution times for streaming multimedia applications on a frame-by-frame basis. Our model is

based on the change-point detection theory used for ATM traffic detection among other applications [82]. We compare our model to prefect prediction and to the exponential moving average used in [62]. The prediction algorithm developed is then used as a part of a power control strategy that merges DVS and DPM.

A second contribution is to merge the DPM and the DVS approaches, by expanding the active state definition to include multiple settings of frequency and voltage, thus resulting in a range of performance and power consumptions available for tradeoff at run time. In this way, the power manager can control performance and power consumption levels both by using DVS when the system is active, and by transitioning components into low-power states when the system is idle.

1.5.3 Energy Efficient System Level Design

The distinctive features of my approach are the following: (i) complete system level and component energy consumption estimates as well as battery lifetime estimates (ii) ability to explore multiple architectural alternatives and (iii) easy estimation of the impact of software changes both during and after the architectural exploration. The tool set is integrated within the *instruction set simulator* (ISS) provided by ARM Ltd. [1]. It consists of two components: a cycle-accurate system level energy consumption simulator with battery lifetime estimation and a system profiler that correlates both energy consumption and performance with the code. My tools have been tested on a real-life industrial application, and have proven to be both accurate (within 5% of hardware measurements) and highly effective in optimizing the energy consumption in embedded systems (energy consumption reduced by 77%). In addition, they are very flexible and easy to adapt to different systems. The tools contain general models for all typical embedded system components but the microprocessor. In order to adopt the tools to another processor, the ARM ISS needs to be replaced by the ISS for the processor of interest.

Energy Efficient Hardware Design

In contrast to previous approaches, in this work memory models and processor instruction level simulator are tightly integrated together with an accurate battery model into cycle-accurate simulation engine. Estimation results obtained with the simulator are shown to be within 5% of measured energy consumption in hardware. In addition, the simulator accurately models the battery discharge current. Since it has only one simulation engine, there is no overhead in executing simulators at different levels of abstraction, or in the interface between them. Thus, this approach enables fast and accurate architecture exploration for both energy consumption and performance.

The tool has been used to redesign the hardware architecture of the SmartBadge. Initially the SmartBadge could execute MPEG video decode in seconds per frame. After the hardware redesign, real time performance with a large reduction in energy consumption is obtained. The tool presented in this thesis enabled fast evaluation of many different memory and processor architectures, together with identifying the appropriate battery and the DC-DC converter. In addition, peak energy consumption due to large switching activity at the CPU-memory bus has been significantly reduced.

Energy Efficient Software Design

Leveraging the estimation engine, I implemented a code profiling tool that gives percentages of time and energy spent in each procedure for every system component. Thanks to energy profiling, the programmer can easily identify the most energy-critical procedures, apply transformations and estimate their impact not only on processor energy consumption, but also on memory hierarchy and system busses.

Finally, with the simulator and the profiler, I developed a code transformation methodology that enables energy (and performance) optimization of software. The methodology consists of three categories of source code optimizations: algorithmic changes, data representation changes and instruction-level optimizations. In addition to a general methodology, processor specific optimizations are introduced that can be used to reduce energy consumption.

1.5.4 Summary of the thesis contribution

The main contributions of this thesis are in the areas of energy efficient system design and utilization. The system utilization is addressed using two different approaches: dynamic power management and dynamic voltage scaling. I also developed modular methodologies for design and simulation of hardware and software energy consumption at the system level.

DPM algorithms reduce the energy consumption at the system level by placing idle components into low-power states. Two algorithms are presented that are based on stationary stochastic models. They are event-driven and give optimal results verified by measurements. The measured power savings range between a factor of 1.7 up to 5.0 with performance basically unaffected. The main limitation of the algorithms is that they require stationary distributions. This problem can be addressed by using adaptive approach such as the one presented in [14].

The DPM model has been extended with a DVS algorithm that reduces energy consumption by changing processor speed and voltage at run-time depending on the needs of the applications running. With a combined approach, savings of a factor of three in energy consumption have been observed. The DVS algorithm assumes non-stationary exponential distributions for the active states, and requires a discrete set of active states. An extension to this work would generalize from the exponential distribution to a general distribution. In addition, with an analog DC-DC converter, it is possible to have a continuous range of active states.

In addition to addressing the utilization issues for energy efficiency, I also present a modular approach for hardware and software design. This methodology is centered around a cycle-accurate simulator of energy dissipation in systems. A profiler that relates energy consumption obtained from the simulator to the source code has been developed. The simulator and the profiler are used together with the software design methodology to guide the design of both energy efficient hardware and software. Energy savings of as much as 77% with performance increase of 92% have been achieved. As the simulator consists of high-level models for components, even more accurate results could be obtained with more detailed component models. In addition, the software optimization methodology is currently manual. Automating some of these optimizations via energy-aware compiler

would speed up the software design.

1.6 Outline of the following chapters

The main contributions of my work are discussed in the rest of this thesis. Chapter 2 presents the dynamic power management algorithms, while Chapter 3 discusses the dynamic voltage scaling algorithm. These two chapters address energy efficient system utilization issues. Energy efficient system design is addressed in Chapter 4 for hardware and in Chapter 5 for software. Finally, the main contributions of this thesis are summarized in Chapter 6.

Chapter 2

Dynamic Power Management

2.1 Introduction

Dynamic power management (DPM) techniques achieve energy efficient utilization of systems by selectively placing system components into low-power states when they are idle. The basic assumption of DPM is that systems and their components experience non-uniform workloads whose variations can be predicted with some accuracy. A power managed system contains a power manager (PM) that implements a control procedure (or policy) based on observations of the workload. Policies can be implemented by different means, such as a timer, a hardware controller or in software.

A power managed system can be modeled as a *power state machine*, where each state is characterized by the power consumption and performance. In addition, state transitions have power and delay cost. Usually, lower power consumption in a given state also implies lower performance and longer transition delay. For example, a hard disk system can be characterized with three states: active, where the disk can read and write, idle state that can transition back to active immediately, and the sleep state that has a significant performance penalty for the transition back into the active state.

When a component is placed into low-power state, it is unavailable for the time period spent in the low-power state, in addition to the transition time between the active and the low-power state. The break-even time, T_{be}, is the minimum time a component should spend in the low-power state to compensate the transition cost. The break-even time can be

calculated directly from the power state machine of the component. A component should be placed into low-power state only when the time spent in that state is longer than the break-even time. With components where the transition cost into inactive state is minimal, the power management policy is trivial (once in the idle state, shut off). In all other situations it is critical to determine the most appropriate policy that the PM will implement. PM policies can be classified into predictive, adaptive and stochastic.

Predictive techniques use past history of the workload in order to predict future idle periods. The goal is to predict when the idle period will be longer than the break-even time, so that the component can be placed into low-power state. The simplest form of predictive techniques are timeout policies. Timeouts assume that the component is very likely to remain idle if it has already been idle for the timeout time. When the timeout is set to the break-even time of the component, the corresponding policy guarantees that the energy consumption will be at worse twice the energy consumed by an ideal policy [43]. The drawback of the timeout policies is that they waste power while waiting for the timeout to expire. Some predictive shut-down policies improve upon timeouts by transitioning the component into low-power state as soon as a new idle period starts. Other predictive policies perform predictive wakeup when they expect the idle time to expire. The main issue with all predictive policies is the quality of prediction of the length and the timing of the idle period. Since the workload is usually unknown a priori and is often non-stationary, it is critical to be able to adapt to the changes in the workload. Several adaptive techniques have been proposed to deal with non-stationary workloads [18, 31]. These techniques are primarily concerned with adjusting the value of the timeout using heuristic measures.

Predictive and adaptive algorithms are heuristic in nature and thus their optimality can only be gauged through comparative simulation and measurement. They assume deterministic response and transition times for the system, and typically only consider two-state systems. In addition, predictive algorithms minimize power and do not control the performance penalty. The stochastic control approaches formulate policy optimization as an optimization problem under uncertainty.

Power management optimization problem can be formulated with the aid of controlled Markov and renewal processes. Using these models it is possible to obtain a globally optimal solution of the performance-constrained power optimization problem that exploits

multiple inactive states of multiple interacting resources under uncertainty. The problem of finding a policy that minimized power consumption under given performance constraints (or vice versa) can be cast as a linear program whose solution is a stationary and randomized policy. Such a policy associates a probability with each command. The command to be issued is selected by a random trial based on the state-dependent probabilities. The policy optimization for both Markov and renewal processes is exact and computationally efficient since the solution is guaranteed to be globally optimal and can be solved in polynomial time. One limitation of these techniques is that complete knowledge of the stochastic characteristics of the system and its workload statistics is assumed. Even though it is possible to construct a model for the system once for all, the workload is often non-stationary and thus much more difficult to characterize in advance.

In this chapter I introduce two new models for power management at the system level that enable modeling system transitions with general distributions, but are still event driven and guarantee optimal results. I implemented the power management algorithms on laptop wireless local area network (WLAN) card and hard disk, desktop hard disk and the SmartBadge [51]. For each of these devices, I collected a set of traces that model typical user behavior well. I found the workload request interarrival times are best modeled with a non-exponential distribution (a Pareto distribution shows the best fit, although the model applies to any distribution or direct data). These results are consistent with the observations on network traffic interarrival times presented in [59]. In addition, I measured the distributions of transition times between active, idle and low power states for each of the system components and found non-exponential transition times into or out of a low power state. Traditional Markov chain models presented in previous work do not apply to these devices since user request arrivals and the transition times of a device are best modeled with the non-exponential distributions. As a result, I formulated the policy optimization problem using two different stochastic approaches.

The first approach is based on renewal theory [77, 67]. It is more concise, but also is limited to systems that have only one decision state. The second approach is based on Time-Indexed Semi-Markov Decision Process model (TISMDP). This model is more general but also more complex. In both cases, the policy optimization problem can be solved *exactly* and in polynomial time by solving a linear program. Both approaches guarantee

optimal solutions. Note that both approaches can handle general user request interarrival distributions, even though in the particular examples presented in this work we use the Pareto distribution since it showed a good fit to the data collected experimentally. The policy decisions are made only upon request arrival or upon finishing serving a request, instead of at every time increment as in discrete-time model. Since policy decisions are made in event-driven manner, more power is saved by not forcing policy re-evaluations as in discrete-time models.

I also present simulation and, more importantly, measurement results on real hardware. The results show that the reduction in power can be as large as 2.4 times with a small performance penalty when power managing the laptop hard disk and 1.7 times for the desktop hard disk. The algorithms perform better than any other power management algorithms tested in [48]. The measurements of optimal policy implemented on a laptop for the WLAN card show that reduction in power can be as large as a factor of 5 with a small performance penalty. Finally, power management results on the SmartBadge show savings of as much as 70% in power consumption.

The remainder of this chapter is organized as follows. Section 2.2 describes the stochastic models of the system components based on the experimental data collected. I develop the model for power management based on renewal theory in Section 2.3. Next, I present the Time-Indexed Semi-Markov Decision Process model for the dynamic power management policy optimization problem in Section 2.4. I show simulation results for the SmartBadge, measured results for power managing WLAN card on a laptop and both simulated and measured results for power managing a hard disk on a laptop and a desktop running Windows OS in Section 2.5.

2.2 System Model

The system can be modeled with three components: the user, device and the queue as shown in Figure 2.1. While the methods presented in this work are general, the optimization of energy consumption under performance constraints (or vice versa) is applied to and measured on the following devices: WLAN card [50] on the laptop, the SmartBadge [51] and laptop and desktop hard disks. The SmartBadge is used as a personal digital assitant

(PDA). The WLAN card enables internet access on the laptop computer running Linux operating system. The hard disks are both part of Windows machines, one in the desktop and the other in the laptop. The queue models a memory buffer associated with each device. In all examples, the user is an application that accesses each device by sending requests via operating system.

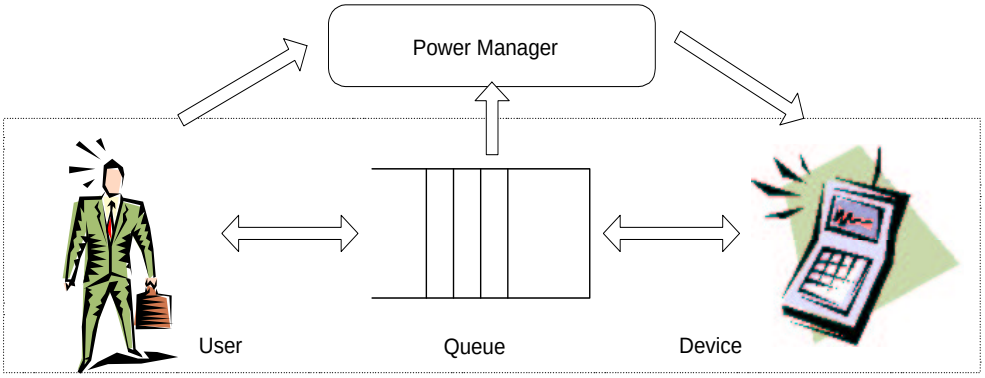

Figure 2.1: System Model

The power management aims at reducing energy consumption in systems by selectively placing components into low power states. Thus, at run time, the power manager (PM) observes user request arrivals, the state of the device's buffer, the power state and the activity level of the device. When all user requests have been serviced, the PM can choose to place the device into a low power state. This choice is made based on a policy. Once the device is in a low power state, it returns to active state only upon arrival of a new request from a user. Note that a user request can come directly from a human user, from the operating system, or even from another device.

Each system component is described probabilistically. The user behavior is modeled by a request interarrival distribution. Similarly, the service time distribution describes the behavior of the device in the active state. The transition distribution models the time taken by the device to transition between its power states. Finally, the combination of interarrival time distribution (incoming jobs to the queue) and service time distribution (jobs leaving the queue) appropriately characterizes the behavior of the queue. These three categories of distributions completely characterize the stochastic optimization problem. The details of each system component are described in the next sections.

2.2.1 User Model

As the user's stochastic model is defined by the request interarrival time distribution, it is of critical importance to collect a set of traces that do a good job at representing the typical user behavior. I collected an 11hr user request trace for the PC hard disks running a Windows operating system with standard software (e.g Excel, Word, Visual C++). In the case of the SmartBadge, I monitored the accesses to the server during multiple long sessions. For the WLAN card we used the *tcpdump* utility [40] to get the user request arrival times for two different applications (telnet and web browser).

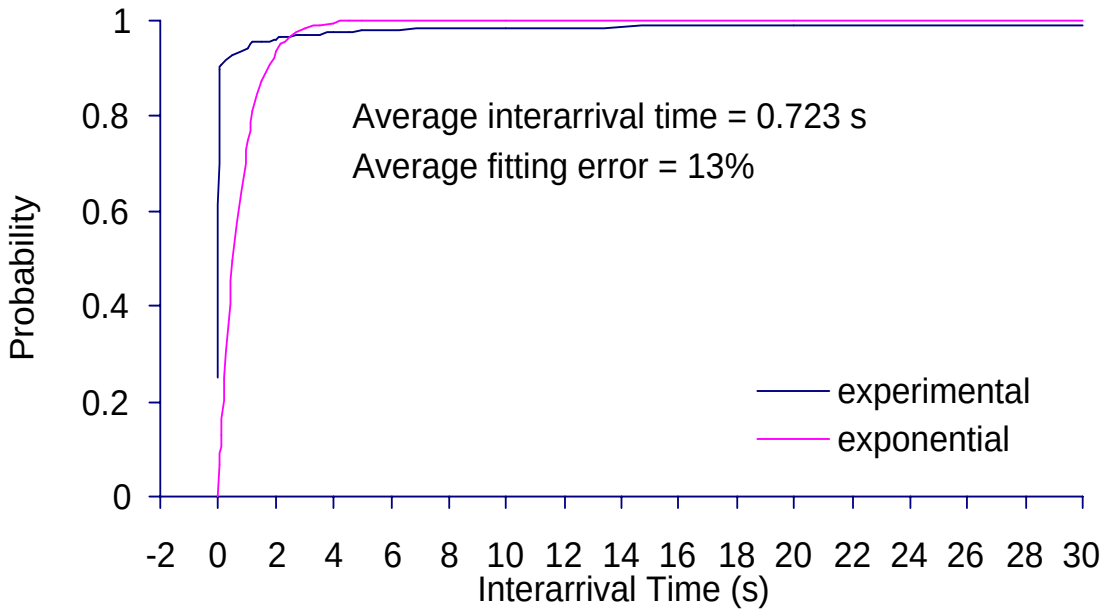

Figure 2.2: User request arrivals in active state for hard disk

The request interarrival times in the active state (the state where one or more requests are in the queue) for all three devices are exponential in nature. Figure 2.2 shows the exponential cumulative distribution fitted to measured results of the hard disk. Similar results have been observed for the other two devices in the active state. Thus, we can model the user in active state with rate λ_U and the mean request interarrival time $\frac{1}{\lambda_U}$ where the probability of the hard disk or the SmartBadge receiving a user request within time

interval t follows the cumulative probability distribution shown below.

$$F_U(t) = 1 - e^{-\lambda_U t} \tag{2.1}$$

The exponential distribution does not model well arrivals in the idle state. The model we use needs to accurately describe the behavior of long idle times as the largest power savings are possible over the long low-power periods. We first filter out short user request interarrival times in the idle state in order to focus on the longer idle times. The filter interval is based on the particular device characteristics and not on the pattern of user access to the device. The filter interval is defined as a fraction of the *break-even* time of the device. Break-even time is the time the device has to stay in the low-power state in order to recuperate the cost of transitioning to and from the low-power state. Transitioning into a low-power state during idle times that are shorter than the break-even time is guaranteed to waste power. Thus it is desirable to filter out very short idle times. We found that filter intervals from 0.5s to about 2s are most appropriate to use for the hard disk, while for the SmartBadge and the WLAN card filter intervals are considerably shorter (50-200ms) since these devices respond much faster than the hard disk.

We use the tail distribution to highlight the probability of longer idle times that are of interest for power management. The tail distribution provides the probability that the idle time is greater than t. Figure 2.3 shows the measured tail distribution of idle periods fitted with the Pareto and the exponential distributions for the hard disk and Figure 2.4 shows the same measurements for the WLAN card. The Pareto distribution shows a much better fit for the long idle times as compared to the exponential distribution. The Pareto cumulative distribution is defined in Equation 2.2. The Pareto parameters are $a = 0.9$ and $b = 0.65$ for the hard disk, $a = 0.7$ and $b = 0.02$ for WLAN web requests and $a = 0.7$ and $b = 0.06$ for WLAN telnet requests. SmartBadge arrivals behave the same way as the WLAN arrivals.

$$F_U(t) = 1 - at^{-b} \tag{2.2}$$

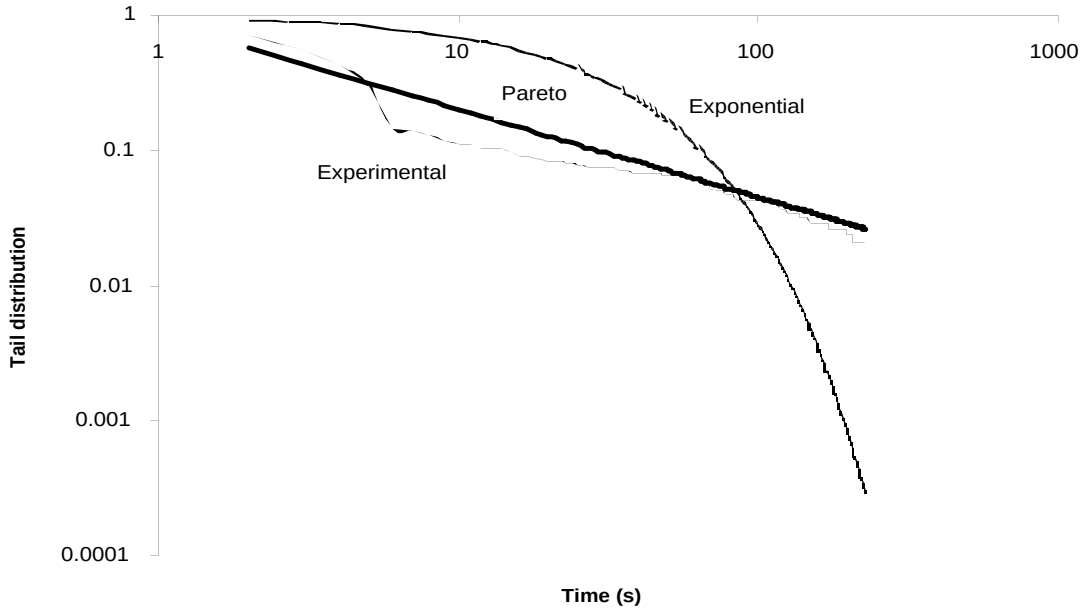

Figure 2.3: Hard disk idle state arrival tail distribution

2.2.2 Portable Devices

Power managed devices typically have multiple power states. Each device has one active state in which it services user requests, and one or more low-power states. The power manager can trade off power for performance by placing the device into low-power states. Each low power state can be characterized by the power consumption and the performance penalty incurred during the transition to or from that state. Usually higher performance penalty corresponds to lower power states.

SmartBadge

SmartBadge and its components are shown in in Figure 1.2. It supports three lower power states: idle, standby and off. The idle state is entered immediately by each component in the system as soon as that particular component is not accessed. The standby and off state transitions can be controlled by the power manager. The transition from standby or off state into the active state can be best described using the uniform probability distribution.

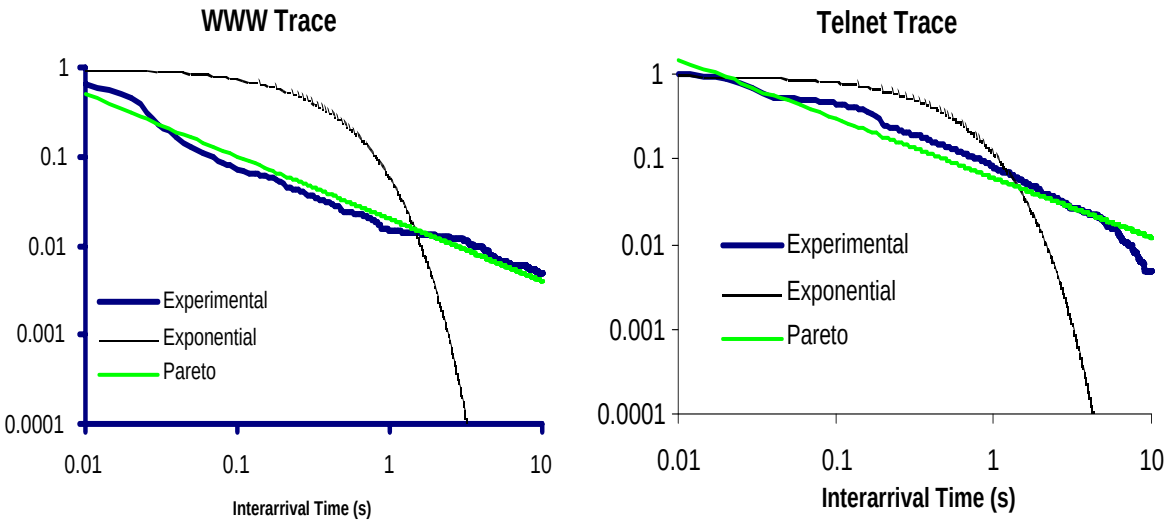

Figure 2.4: WLAN idle state arrival tail distribution

Components in the SmartBadge, the power states and the transition times of each component from standby (t_{sby}) and off (t_{off}) state into active state, and the transition times between standby and off states (t_{so}) are shown in Table 2.1. Note that the SmartBadge has two types of data memory – slower SRAM (1MB, 80ns) from Toshiba and faster DRAM (4MB, 20ns) from Micron that is used only during MPEG decode. Memory takes longer to transition from off to active state as contents of RAM have to be downloaded from FLASH and initialized. The power consumption of all components in the off state is $0mW$.

Table 2.1: SmartBadge components

Component	Active Pwr (mW)	Idle Pwr (mW)	Standby Pwr (mW)	t_{sby} (ms)	t_{off} (ms)	t_{so} (ms)
Display	1000	1000	100	100	240	110
RF Link	1500	1000	100	40	80	20
SA-1100	400	170	0.1	10	35	10
FLASH	75	5	0.023	0.6	160	150
SRAM	115	17	0.13	5.0	100	90
DRAM	400	10	0.4	4.0	90	75
Total	3.5 W	2.2 W	200 mW	110 ms	705 ms	455 ms

WLAN card

The wireless card has multiple power states: two active states, *transmitting*, *receiving*, and two inactive states, *doze* and *off*. Transmission power is 1.65W, receiving 1.4W, the power consumption in the doze state is 0.045W [50] and in the off state it is 0W. When the card is awake (not in the off state), every 100ms it synchronizes its clock to the access point (AP) by listening to the AP beacon. After that, it listens to the TIM map to see if it can receive or transmit during that interval. Once both receiving and transmission are done, it goes into the doze state until the next beacon [19]. This portion of the system is fully controlled from the hardware and thus is not accessible to the power manager that has been implemented at the OS level.

The power manager can control the transitions between the doze and the off states. Once in the off state, the card waits for the first user request arrival before returning back to the doze state. We measured the transitions between the doze and the off states using *cardmgr* utility. The transition from the doze state into the off state takes on average $t_{ave} = 62ms$ with variance of $t_{var} = 31ms$. The transition back takes $t_{ave} = 34ms$ with $t_{var} = 21ms$ variance. The transition between doze and off states are best described using the uniform distribution.

Hard Disks

We considered two different hard disks in our experiments: the Fujitsu MHF 2043AT hard disk in the Sony Vaio laptop and the IBM hard disk in VAResearch desktop. Service times on the hard disks in the active state most closely follow an exponential distribution as shown in Figure 2.5. We found similar results for the SmartBadge and the WLAN card. The average service time is defined by $\frac{1}{\lambda_D}$ where λ_D is the average service rate. Equation 2.3 defines the cumulative probability of the device servicing a user request within time interval t.

$$F_D(t) = 1 - e^{-\lambda_D t} \tag{2.3}$$

The power manager can control the transitions between the idle and the sleep state on both of the hard disks. The power consumptions in the idle and sleep states, in addition

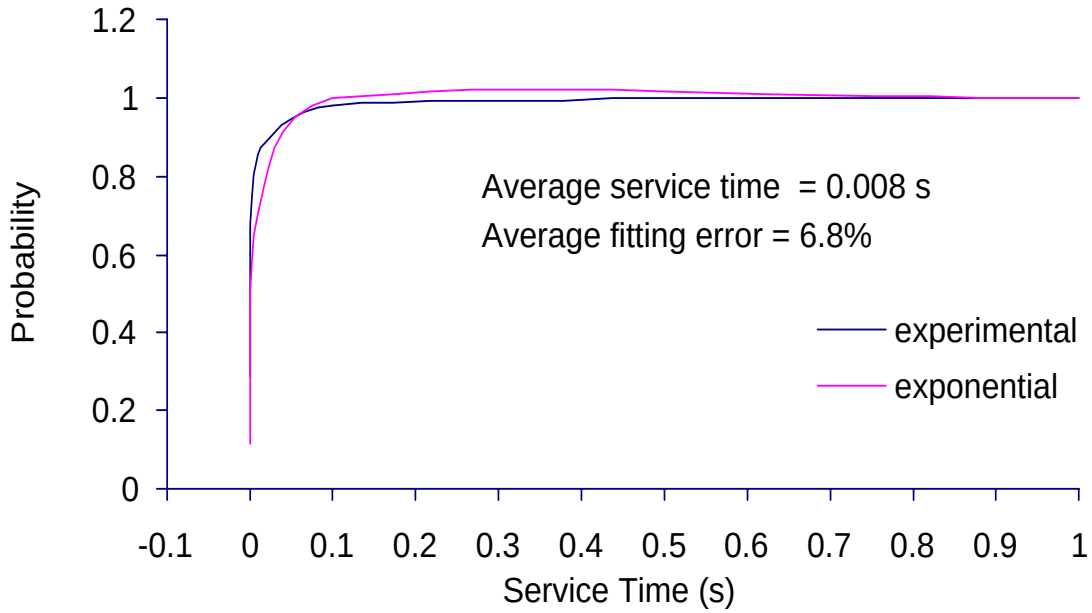

Figure 2.5: Hard disk service time distribution

to the average transition times between those states are shown in Table 2.2. The transition from sleep to active state requires spin-up of the hard disk, which is very power intensive. While in the sleep state, the disk consumes much less power.

Model	P_{sleep} Watt	P_{active} Watt	T_{sleep} sec	T_{active} sec
IBM	0.75	3.48	0.51	6.97
Fujitsu	0.13	0.95	0.67	1.61

Table 2.2: Disk Parameters

Once in the sleep state, the hard disk waits for the first service request arrival before returning to the active state. The transition between active and sleep states is best described using the uniform distribution, where t_0 and t_1 can be defined as $t_{ave} - \Delta t$ and $t_{ave} + \Delta t$ respectively. The cumulative probability function for the uniform distribution is shown

below.

$$F_D(t) = \begin{cases} 0 & t \leq t_0 \\ \frac{t-t_0}{t_1-t_0} & t_0 < t \leq t_1 \\ 1 & t > t_1 \end{cases} \tag{2.4}$$

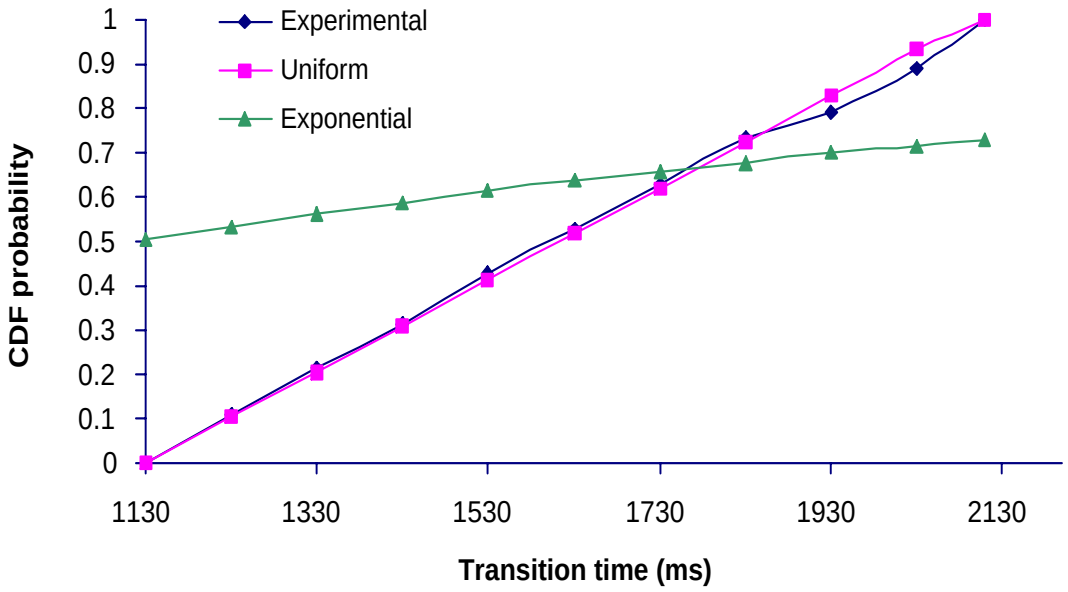

Figure 2.6: Hard disk transition from sleep to active state

Figure 2.6 shows the large error that would be made if the transition to the sleep state were approximated using an exponential distribution. For example, the transition for Fujitsu hard disk from the active state into the sleep state takes on average $0.67s$ with variance of $0.1s$. The transition back into the active state is much longer, requiring $1.6s$ on average with $0.5s$ variance.

2.2.3 Queue

Portable devices normally have a buffer for storing requests that have not been serviced yet. Since we did not have access to the detailed information about the real-time size of each

queue, we measured the queue size of maximum 10 jobs with an experiment on a hard disk using a typical user trace. Because the service rate in the SmartBadge and WLAN card is higher, and the request arrival rate is comparable, we assume that the same maximum queue size can be used. As the requests arriving at the hard disk do not have priority associated with them, and the SmartBadge requests by definition do not have priority, our queue model contains only the number of jobs waiting for service. Active and low-power states can be differentiated then by the number of jobs pending for service in the queue.

2.2.4 Model Overview

Table 2.3 shows the probability distributions used to describe each system component derived from the experimental results. User request interarrival times with at least one job in the queue are best modeled with the exponential distribution. On the other hand, we have shown that in all four applications, the Pareto distribution is best used to model the arrival of the user's requests when the queue is empty. Note that the queue is empty in either idle state or a low power state. The device is in the active state when at least one job is waiting to be serviced. We have also shown that the service times in the active state are best modeled with the exponential distribution. The transitions to and from the low power states are better modeled with a uniform distribution. The combination of these distributions is used to derive the state of the queue. Thus in the active state two exponential distributions define the number of jobs in the queue: the interarrival time and the service time distributions. During transitions, the queue state is defined by the transition distribution and the distribution describing user request arrivals. During transitions and in the low-power states the first arrival follows the Pareto distribution, but the subsequent arrivals are modeled with the exponential distribution since for very short interarrival times the exponential distribution is very close to the Pareto distribution and the experimental results, as can be seen in Figures 2.3 and 2.4.

Although in the experimental section of this paper we utilize the fact that the non-exponential user and device distributions can be described with well-known functions (Pareto or uniform), the models we present in the following sections are general in nature and thus can give optimal results with both experimental distributions obtained at run

Table 2.3: System Model Overview

System Component	Component State	Distribution
User	Queue not empty	Exponential
	Queue empty	Pareto
Device	Active	Exponential
	Transition	Uniform

time or commonly used theoretical distributions. We found that in the particular examples we present in this work the Pareto, and the uniform distributions enabled us to obtain the optimal policy faster without sacrificing accuracy.

2.3 DPM Based on Renewal Theory

Renewal theory [77, 67] studies stochastic systems whose evolution over time contains a set of *renewals or regeneration times* where the process begins statistically anew. Formally, a renewal process specifies that the random times between system renewals be independently distributed with a common distribution $F(x)$. Thus the expected time between successive renewals can be defined as:

$$E[\tau] = \int_0^\infty x dF(x) \tag{2.5}$$

Note that the Poisson process is a simple renewal process for which renewal times are distributed with the exponential distribution. In this case, the common distribution between renewals can be defined as $F(x) = 1 - e^{-\lambda x}$, and the mean time between renewals (or between the exponential arrivals) is defined as $E[\tau] = 1/\lambda$. A process can be considered to be a renewal process only if there is a state of the process in which the whole system probabilistically restarts. This, of course, is the case in any system that is completely described by the exponential or the geometric distributions, since those distributions are not history dependent (they are memoryless).

In policy optimization for dynamic power management, the complete cycle of transition

from the idle state, through the other states and then back into the idle state can be viewed as one renewal of the system. When using renewal theory to model the system, the decision regarding transition to a lower power state (e.g. the sleep state) is made by the power manager in the idle state. If the decision is to transition to the lower power state, the system re-enters the idle state after traversing through a set of states. Otherwise, the system transitions to the active state on the next job arrival, and then returns to the idle state again once all jobs have been serviced.

The general system model shown in Figure 2.1 defines the power manager (PM), and three system components: user, device and the queue. To provide concreteness in our examples, each component is completely specified by the probability distributions defined in the previous section. With renewal theory, the search for the best policy for a system modeled using stationary non-exponential distributions can be cast into a stochastic control problem.

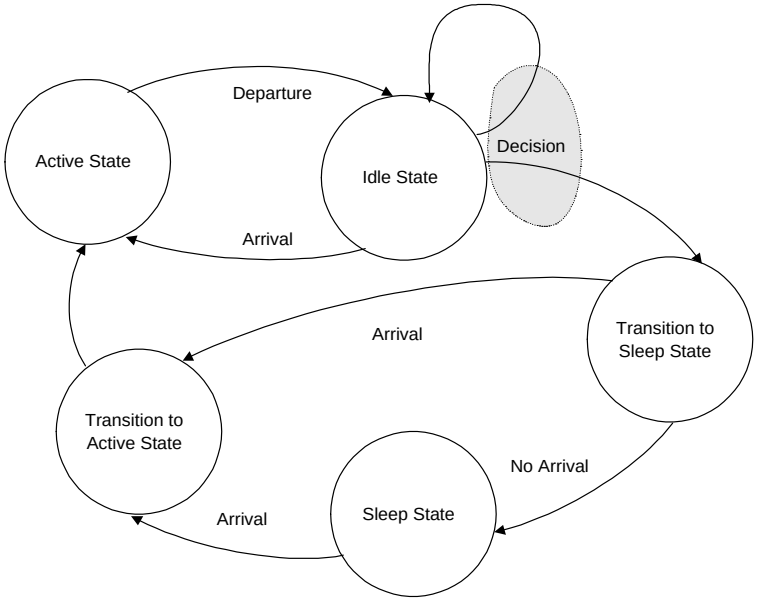

Figure 2.7: System states for renewal theory model

System states used in the formulation of the renewal model are shown in Figure 2.7. In the active state the queue contains at least one job pending and the request arrivals and service times follow the exponential distributions. Once the queue is emptied, the system

transitions to the idle state, which is also the renewal and decision point in this system. Upon arrival of request, the system always transitions back into the active state. The PM makes a decision on when the transition to a low-power state from the idle state should occur. As soon as the command to place the system into the low-power state is given, the system starts a transition between the idle and the low-power states. The transition state highlights the fact that device takes a finite and random amount of time to transition into the low power state (governed by a uniform distribution). If during the transition time a request arrives from the user (first request follows the Pareto distribution, subsequent requests are exponential), the device starts the transition to active state as soon as the transition to off state is completed. If no request arrives during the transition state, the device stays in a low-power state until the next request arrives (the Pareto distribution). Upon request arrival, the transition back into the active state starts. Once the transition into the active state is completed, the device services requests, and then again returns to the idle state where the system probabilistically renews again.

2.3.1 Renewal Theory Model

We formulate the power management policy optimization problem based on renewal theory in this section. We use upper-case bold letters (*e.g.*, **M**) to denote matrices, lower-case bold letters (*e.g.*, **v**) to denote vectors, calligraphic letters (*e.g.*, $\mathcal{S}$) to denote sets, upper-case letters (*e.g.*, S) to denote scalar constants and lower-case letters (*e.g.*, s) to denote scalar variables.

The problem of power management policy optimization is to determine the optimal distribution of the random variable Γ that specifies when the transition from the idle state to low-power state should occur based on the last entry into the idle state. We assume that Γ takes on values in $[0, h, 2h, \ldots, jh, \ldots, Nh)$, where j is an index, h is a fraction of the break-even time of the device, and N is the maximum time before the system goes to a low-power state (usually set to an order of magnitude greater than break-even time). The solution to the policy optimization problem can be viewed as a table of probabilities (Γ), where each element $p(j)$ specifies the probability of transition from idle to a low-power state indexed by time values jh.

We can formulate an optimization problem to minimize the average performance penalty under the power constraint ($P_{constraint}$) using the results of the ratio-limit theorem for renewal processes [67], as shown in Equation 2.6. The average performance penalty is calculated by averaging $q(j)$, the time penalty user incurs due to transition to low-power state, over, $t(j)$, the expected time until renewal. The power constraint is shown as an equality as the system will use the maximum available power in order to minimize the performance penalty. The expected energy ($\sum_j p(j)e(j)$) is calculated using $p(j)$, the probability of issuing command to go to low-power state at time jh, and $e(j)$, the expected energy consumption. This expected energy has to equal the expected power constraint ($\sum_j p(j)t(j)P_{constraint}$) calculated using $t(j)$, the expected time until renewal, $P_{constraint}$, the power constraint, and $p(j)$. The unknown in the optimization problem is $p(j)$, the probability of issuing a command to go to low-power state at time jh. The full derivation of all the quantities follows.

$$\begin{aligned} \min\ & \frac{\sum_j p(j)q(j)}{\sum_j p(j)t(j)} \\ \text{s.t.}\ & \sum_j p(j)[e(j) - t(j)P_{constraint}] = 0 \\ & \sum_j p(j) = 1 \\ & p(j) \geq 0 \quad \forall j \end{aligned} \tag{2.6}$$

Computation of Renewal Time

Given the state space shown in Figure 2.7, we can define the expected time until renewal, $t(j)$, as follows. We define β as the time at which the first job arrives after the queue has been emptied. The first arrival is distributed using general probability distribution, $P(jh)$. We also define the indicator function, I(jh), that is equal to one if we are in interval jh and is zero otherwise.

Further, as we showed in Section 2.2, the subsequent user request arrivals follow a Poisson process with rate λ_U. Finally, the servicing times of the device also can be described using the exponential distribution with parameter λ_D. We can now define the expected time until renewal (τ_j) for each time increment spent in the idle state as sum of expected time

until renewal if arrival comes before the system starts transitioning into low-power state jh (as shown by the first cycle in Figure 2.8 and the first half of Equation 2.7) and if the arrival comes after the transitions has already started (the second parts of Figure 2.8 and Equation 2.7).

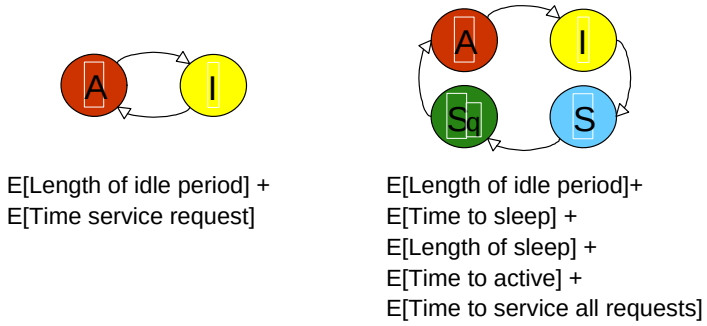

Figure 2.8: Renewal Cycles

$$t(j) = E[\tau_j I(\beta \leq jh) | \Gamma = jh] + E[\tau_j I(\beta > jh) | \Gamma = jh] \tag{2.7}$$

Each of the two terms in Equation 2.7 is defined in Equations 2.8 and 2.10. Note that Figure 2.8 shows the components of each of the two terms. The expected time until renewal for arrival coming before transitioning to low-power state at time jh (the left portion of Figure 2.8) is the expected time until arrival (the first term in Equation 2.8) and the time needed to work off the request that just arrived (the second term). Note that the second term is based on the results from M/M/1 queueing theory due to the fact that the time to work off the request is governed by the exponential distribution with rate λ_D, while the arrivals in the active state are described by the exponential distribution with rate λ_U. The job departure rate has to be larger than the arrival rate ($\lambda_D \geq \lambda_U$), otherwise the queue would overflow. In all cases we studied, λ_D is at least order of magnitude larger than λ_U, leading to:

$$E[\tau_j I(\beta \leq jh) | \Gamma = jh] = \sum_{k=1}^{j} khP(\beta = kh) + \frac{1}{\lambda_D - \lambda_U} P(\beta \leq jh) \tag{2.8}$$

If the arrival comes after time jh when the system starts the transition to low-power state (the right portion of Figure 2.8), then the expected time until renewal is the sum of the time

until arrival (jh), with expected times for transition to low-power state and back to active states (EU_1, EU_2), expected length of the low-power period and the expected time to work off requests that arrived during the renewal period:

$$E[\tau_j I(\beta > jh) | \Gamma = jh] = \tag{2.9}$$
$$P(\beta > jh) \left[\begin{array}{l} jh + EU_1 + EU_2 + \\ E[(\beta - jh + EU_1) I(\beta > jh + EU_1)] + \\ E[(jh + EU_1 - \beta) I(jh < \beta \leq jh + EU_1)] \frac{\lambda_D}{\lambda_D - \lambda_U} + \\ \frac{1}{\lambda_D - \lambda_U} + EU_2 \frac{\lambda_D}{\lambda_D - \lambda_U} \end{array} \right]$$

Computation of Costs

We can define the performance penalty that the user experiences due to transition to low power state ($q(j)$) and the expected energy consumption ($e(j)$) for each state using the same set of equations, just with different values for constants (c) as shown in Table 2.4. Each state is labeled on the left side, while the expected time spent in that state multiplied by the constant c is on the right side.

The constants (c) equal the power consumption in a given state for energy consumption computation. For the performance penalty the constants should be set to zero in low-power state and idle state and to one in all other states. For example, the constant c_i is set to power consumption of the device while in the idle state when calculating energy consumption (the first equation). Since there is no performance penalty to servicing users requests in the idle state, the constant c_i is set to zero for performance penalty calculation. On the other hand, the transition to the active state causes performance degradation, thus the constant c_{ta} is here set to one. The same constant is set to power required for the transition to the active state when calculating energy consumption.

The expected times spent in each state outlined in Table 2.4 are calculated as follows:

- **Idle State:** The expected time spent in the idle state is the expected average of the idle time until the first request arrival ($\sum_{k=0}^{j} khP(\beta = kh)$) and the time spent in the idle state when the transition to low power state occurs before the first arrival ($jhP(\beta \geq jh)$).

Table 2.4: Calculation of Costs

State	Performance penalty or Energy consumption
Idle	$c_i[\sum_{k=0}^{j} khP(\beta = kh) + jhP(\beta \geq jh)]$
To Low Power	$c_{ts}[EU_1 P(\beta \geq jh)]$
Low Power	$c_s[E[\beta - (jh + EU_1)]I(\beta > jh + EU_1)P(\beta \geq jh)]$
To Active	$c_{ta}[EU_2 P(\beta \geq jh)]$
Active	$c_a[\frac{1}{\lambda_D - \lambda_U} P(\beta < jh) +$ $EU_2 \frac{\lambda_D}{\lambda_D - \lambda_U} P(\beta \geq jh) +$ $\frac{1}{\lambda_D - \lambda_U} P(\beta \geq jh) +$ $E[(jh + EU_1 - \beta)I(jh \leq \beta \leq jh + EU_1)] \frac{\lambda_D}{\lambda_D - \lambda_U} P(\beta \geq jh)]$

- **Transition To Low Power State:** The transition to low power state occurs only if there has been no request arrival before the transition started ($P(\beta \geq jh)$). The expected average time of the transition to low power state is defined by the average of the uniform distribution that describes the transition (EU_1).

- **Low Power State:** Low power state is entered only if no request arrival occurred while in the idle state ($P(\beta \geq jh)$). The device stays in that state until the first request arrives ($E[\beta - (jh + EU_1)]I(\beta > jh + EU_1)$).

- **Transition To Active State:** The transition to active state occurs only when there is a successful transition to low power state ($P(\beta \geq jh)$). The transition length is the expected average of uniform distribution that describes the transition to active state

(EU_2).

- **Active State:** The device works off the request that arrived in the idle state if no transition to low power state occurred ($\frac{1}{\lambda_D - \lambda_U} P(\beta < jh)$). If the transition to low power state did occur (terms containing $P(\beta \geq jh)$), then the system is in the active state for the time it takes to work off all the requests that arrived while transitioning between idle, low power and active states.

Problem Formulation

The optimization problem shown in Equation 2.6 can be transformed into a linear program (LP) using intermediate variables $y(j) = \frac{p(j)}{\sum_j p(j)t(j)}$ and $z(j) = 1/\sum_j p(j)t(j)$ [10].

$$\begin{aligned} \textbf{LP: } \min & \sum_j q(j)y(j) \\ \text{s.t. } & \sum_j [e(j)y(j) - t(j)z(j)P_{Constraint}] = 0 \\ & \sum_j t(j)y(j) = 1 \\ & z \geq 0 \end{aligned} \tag{2.10}$$

Once the values of intermediate variables $y(j)$ and $z(j)$ are obtained by solving the LP shown above, the probability of transition to low-power state from idle state at time jh, $p(j)$, can be computed as follows:

$$p(j) = \frac{y(j)}{z(j)} \tag{2.11}$$

2.3.2 Policy Implementation

The optimal policy obtained by solving the LP given in Equation 2.10 is a table of probabilities $p(j)$. The policy can be implemented in two different ways. If probability distribution defined by $p(j)$ is used, then on each interval jh the policy needs to be re-evaluated until either a request arrives or the system transitions to a low-power state. This implementation has a high overhead as it requires multiple re-evaluations. An alternative implementation

gives the same results, but it requires only one evaluation upon entry to idle state. In this case a table of cumulative probabilities $P(j)$ is calculated based on the probability distribution described with $p(j)$. Once the system enters the idle state, a pseudo-random number *RND* is generated and normalized. The time interval for which the policy gives the cumulative probability $P(j)$ of going to the low-power state greater than *RND* is the time when the device will be transitioned into the low-power state. Thus the policy works like a randomized timeout. The device stays in the idle state until either the transition to the low-power state as given by *RND* and the policy, or until a request arrival forces the transition into the active state. Once the device is in the low-power state, it stays there until the first request arrives, at which point it transitions back into the active state.

Example 2.3.1 *If a sample policy is given in Table 2.5, and the pseudo-random number RND generated upon entry to idle state is 0.6, then the power manager will give a command to transition the device to the low power state at time indexed by* $j = 3$*. Thus, if the time increment used is 0.1 second, then the device will transition into low power state once it has been idle for 0.3 seconds. If a user request arrives before 0.3 seconds have expired, then the device transitions back to the active state.*

Table 2.5: Sample Policy

Idle Time	Transition Probability
j	P(j)
0	0
1	0.1
2	0.4
3	0.9
4	1.0

2.4 DPM Based on Time-Indexed Semi-Markov Decision Processes

In this section we present the power management optimization problem formulation based on Time-Indexed Semi-Markov Decision Processes. This model is more general than the model based on renewal theory as it enables multiple decision points (see Example 2.4.1). Our goal is to minimize the performance penalty under an energy consumption constraint (or vice versa). We first present the average-cost semi-Markov decision process optimization problem [63] and then extend it to the time-indexed SMDP for modeling general inter-arrival times.

Example 2.4.1 *The SmartBadge has two states where decisions can be made: idle and standby. The idle state has higher power consumption, but also a lower performance penalty for returning to the active state as compared to the standby state. From the idle state, it is possible to give a command to transition to the standby or the off states. From standby, only a command to transition to the off state is possible. The optimal policy determines when the transition between idle, standby and off states should occur.*

At each event occurrence, the power manager issues a *command* (or *action*) that decides the next state to which the system should transition. In general, commands given are functions of the state history and the policy. Commands are modeled by decisions, which can be deterministic or randomized. In the former case, a decision implies issuing a command, in the later case it gives the probability of issuing a command. The decisions taken by the PM form a discrete sequence $[\delta^{(1)}, \delta^{(2)}, \dots)$. The sequence completely describes the PM *policy* π which is the unknown of our optimization problem. Among all policies two classes are particularly relevant, as defined next.

Definition 2.4.1 *Stationary policies are policies where the same decision* $\delta^{(i)} = \delta$ *is taken at every decision point* t_i, $i = 1, 2, \dots$, *i.e.,* $\pi = [\delta, \delta, \dots)$.

For stationary policies, decisions are denoted by δ, which is a function of the system state s. Thus, stationarity means that the *functional dependency* of δ on s does not change over time. When s changes, however, δ can change. Furthermore, notice that even a constant

decision *does not* mean that the *same command* is issued at every decision point. For randomized policies, a decision is a probability distribution that assigns a probability to each command. Thus, the actual command that is issued is obtained by randomly selecting from a set of available commands with the probabilities specified by δ.

Definition 2.4.2 *Markov stationary policies are policies where decisions δ do not depend on the entire history but only on the state of the system s at the current time.*

Randomized Markov stationary policies can be represented as a $S \times A$ decision matrix $\mathbf{P}_\pi$. An element $p_{s,a}$ of $\mathbf{P}_\pi$ is the probability of issuing command *a* given that the state of the system is *s*. *Deterministic* Markov stationary policies can still be represented by matrices where only one element for each row has value 1 and all other elements are zero. The importance of these two classes of policies stems from two facts: first, they are *easy to store and implement*, second, we will show that for our system model, *optimal policies belong to these classes*. In the next sections, we will first present the average cost semi-Markov model (SMDP), followed by the extension to time-indexed SMDP.

2.4.1 Semi-Markov Average Cost Model

Semi-Markov decision processes (SMDP) generalize Markov decision processes by allowing the decision maker to choose actions whenever the system state changes, to model the system evolution in continuous time and to allow the time spent in a particular state to follow an arbitrary probability distribution. *Continuous-time Markov decision processes* [71, 64] can be viewed as a special case of Semi-Markov decision processes in which the inter-transition times are always exponentially distributed. Figure 2.9 shows a progression of the SMDP through event occurrences, called decision *epochs*. The power manager makes decisions at each event occurrence. The *interevent time set* is defined as $\mathcal{T} = \{t_i, \text{ s.t. } i = 0, 1, 2, \ldots, i_{max}\}$ where each t_i is the time between the two successive event arrivals and i_{max} is the index of the maximum time horizon. We denote by $s_i \in \mathcal{S}_i$ the system state at decision epoch i. Commands are issued whenever the system state changes. We denote by $a_i \in \mathcal{A}$ an action that is issued at decision epoch i. When action a_i is chosen in system state s_i, the probability that the next event will occur by time t_i is defined

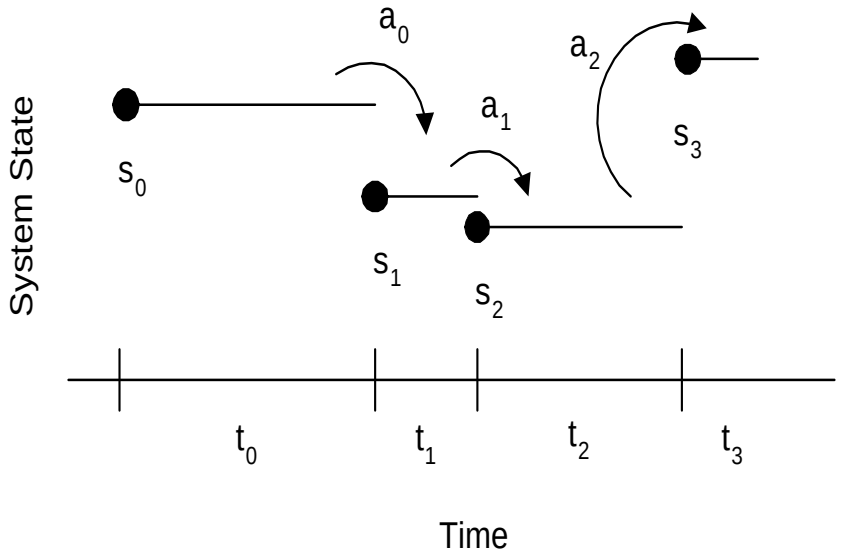

Figure 2.9: SMDP Progression

by the cumulative probability distribution $F(t_i|s_i,a_i)$. Also, the probability that the system transitions to state s_{i+1} at or before the next decision epoch t_i is given by $p(s_{i+1}|t_i,s_i,a_i)$.

The SMDP model also defines cost metrics. The average cost incurred between two successive decision epochs (events) is defined in Equation 2.12 as a sum of the lump sum cost $k(s_i,a_i)$ incurred when action a_i is chosen in state s_i, in addition to the cost in state s_{i+1} incured at rate $c(s_{i+1},s_i,a_i)$ after choosing action a_i in state s_i. We define $\mathcal{S}_{i+1}$ as the set of all possible states that may follow s_i.

$$cost(s_i,a_i) = k(s_i,a_i) + \int_0^{\infty} [F(du|s_i,a_i) \sum_{s_{i+1}\in S_{i+1}} \int_0^{u} c(s_{i+1},s_i,a_i) p(s_{i+1}|t_i,s_i,a_i)]dt \quad (2.12)$$

We can define the total expected cost for policy π until time t as a sum of all lump sum costs $k_v(s,a)$ up to time t and the costs incurred at the rate $c(s,a)$ while in each state s until time t:

$$v_t^{\pi}(s) = E_s^{\pi} \left\{ \int_0^t c(s,a,u)du + \sum_{v=v_0^{\pi}}^{v_{t-1}^{\pi}} k_v(s,a) \right\} \quad (2.13)$$

and then we can define the average expected cost for all s:

$$g^{\pi}(s) = \lim_{t\to\infty} inf \frac{v_t^{\pi}(s)}{t} \tag{2.14}$$

Theorem 2.4.1 *Finding the optimal power management policy π minimizing Equation 2.14 is equivalent to solving the following problem:*

$$h(s) = \min_{a\in A}\{cost(s,a) - g(s)y(s,a) + \sum_{j\in S} m(j|s,a)h(j)\} \tag{2.15}$$

where $h(s)$ is the so called bias (the difference between long term average cost and the average cost per period for a system in steady state [63]), $g(s)$ is the average cost, $m(j|s,a)$, the probability of arriving to state j given that the action a was taken in state s is defined by:

$$m(j|s,a) = \int_0^{\infty} p(j|t,s,a)F(dt|s,a) \tag{2.16}$$

and expected time spent in each state is given by:

$$y(s,a) = \int_0^{\infty} t\sum_{s\in S} p(j|t,s,a)F(dt|s,a) \tag{2.17}$$

Proof of Theorem 2.4.1 is given in [63].

The following examples illustrate how the probability, the expected time and energy consumption can be derived.

Example 2.4.2 *In the active state with at least one element in the queue, we have two exponential random variables, one for the user with parameter λ_U and one for the device with parameter λ_D. The probability density function of the jointly exponential user and device processes defines an M/M/1 queue and thus can be described by $F(dt|s,a) = \lambda e^{-\lambda t}dt$, where $\lambda = \lambda_U + \lambda_D$. In the same way, the probabilities of transition in M/M/1 queue, $p(j|t,s,a)$, are defined as λ_U/λ for request arrival and λ_D/λ for request departure.*

Using Equation 2.16 we derive that the probability of transition to the state that has an additional element in the queue is λ_U/λ , while the probability of transition to the state with one less element is given by λ_D/λ. Note that in this special case $p(j|t,s,a) = m(j|s,a)$. The expected time for transition derived using Equation 2.17 is given by $1/\lambda$, which is again characteristic of M/M/1 queue. Energy consumption is given in Equation 2.12. For this specific example, we define the power consumption in active state with P_a and we assume that there is no fixed energy cost for transition between active states. Then the energy consumption can be computed as follows: $cost(s,a) = \int_0^\infty \lambda e^{-\lambda t} dt [\int_0^t P_a \lambda_D/\lambda du + \int_0^t P_a \lambda_U/\lambda du]$ which is equal to $\frac{P_a}{\lambda}$. Note that this solution is very intuitive, as we would expect the energy consumption to equal the product between the power consumption and the expected time spent in the active state.

The second example considers the transition from the sleep state into the active state with one or more elements in the queue.

Example 2.4.3 *The transition from the sleep to the active state is governed by two distributions. A uniform distribution describes device transitions: $F_U(dt|s,a) = dt/(t_{max_{as}} - t_{min_{as}})$, where $t_{max_{as}}$ and $t_{min_{as}}$ are maximum and minimum transition times. The request arrival distribution is exponential: $F_E(dt|s,a) = \lambda_U e^{-\lambda_U t} dt$. The probability of no arrival during the transition is given by $p(j|t,s,a) = e^{-\lambda_U t}$.*

The probability of transition from the sleep state with a set number of queue elements, into an active state with the same number of elements in the queue is given by: $m(j|s,a) = \int_{t_{min_{as}}}^{t_{max_{as}}} \frac{e^{-\lambda_U t}}{(t_{max_{as}} - t_{min_{as}})} dt$. The expected transition time, $y(s,a)$, is given by $(t_{max_{as}} + t_{min_{as}})/2$, which can be derived with Equation 2.17. Finally, the energy consumed during the transition is defined by $cost(s,a) = \int_0^\infty \frac{du}{t_{max_{as}} - t_{min_{as}}} \int_0^u P_{sa} dt$ assuming that there is no fixed energy consumed during the transition, and that the power consumption for the transition is given by P_{sa}. The energy consumption can further be simplified to be $\frac{2P_{sa}}{t_{max_{as}} + t_{min_{as}}}$. This is again equal to the product of power consumption with the expected transition time from the sleep state into the active state.

The problem defined in Theorem 2.4.1 can be solved using policy iteration or by formulating and solving a linear program. There are two main advantages of linear programming formulation: additional constraints can be added easily, and the problem can be solved in polynomial time (in $S \cdot A$). The primal linear program derived from Equation 2.15 defined in Theorem 2.4.1 can be expressed as follows:

$$\begin{aligned} \textbf{LPP:} \ & \min g(s) \\ & \text{s.t. } g(s)y(s,a) + h(s) - \sum_{j \in S} m(j|s,a)h(j) \geq cost(s,a) \quad \forall s,a \end{aligned} \tag{2.18}$$

where s and a are the state and command given in that state, $g(s)$ is the average cost, $h(s)$ is the bias, $y(s,a)$ is the expected time, $cost(s,a)$ is the expected cost (e.g. energy), and $p(j|s,a)$ is the transition probability between the two states.

Because the constraints of LPP are convex in $g(s)$ and the Lagrangian of the cost function is concave, the solution to the primal linear program is convex. In fact, the constraints form a polyhedron with the objective giving the minimal point within the polyhedron. Thus, the **globally optimal** solution can be obtained that is both stationary and deterministic. The dual linear program shown in Equation 2.19 is another way to cast the same problem (in this case with the addition of a performance constraint). The dual LP shows the formulation for minimizing energy under performance constraint (opposite problem can be formulated in much the same way).

$$\begin{aligned} \textbf{LPD:} \ & \min \sum_{s \in S} \sum_{a \in A} cost_{energy}(s,a) f(s,a) \\ & \text{s.t. } \sum_{a \in A} f(s,a) - \sum_{s' \in S} \sum_{a \in A} m(s'|s,a) f(s',a) = 0 \\ & \sum_{s \in S} \sum_{a \in A} y(s,a) f(s,a) = 1 \\ & \sum_{s \in S} \sum_{a \in A} cost_{perf}(s,a) f(s,a) < Constraint \end{aligned} \tag{2.19}$$

The $A \cdot S$ unknowns in the LPD, $f(s,a)$, called *state-action frequencies*, are the expected number of times that the system is in state s and command a is issued. It has been shown that the exact and the optimal solution to the SMDP policy optimization problem belongs

to the set of Markovian randomized stationary policies [63]. A *Markovian randomized stationary policy* can be compactly represented by associating a value $x(s,a) \leq 1$ with each state and action pair in the SMDP. The probability of issuing command a when the system is in state s, $x(s,a)$, is defined in Equation 2.20.

$$x(s_i,a_i) = \frac{f(s_i,a_i)}{\sum_{a_i \in A} f(s_i,a_i)} \tag{2.20}$$

2.4.2 Time-Indexed Semi-Markov Average Cost Model

The average-cost SMDP formulation presented above is based on the assumption that at most one of the underlying processes in each state transition is not exponential in nature. On transitions where none of the processes are exponential, time-indexed Markov chain formulation needs to be used to keep the history information. Without indexing, the states in the Markov chain would have no information on how much time has passed. As for all distributions, but the exponential, the history is of critical importance, the state space has to be expanded in order to include the information about time as discussed in [77]. Time-indexing is done by dividing the time line into a set of intervals of equal length Δt. The original state space is expanded by replacing one idle and one queue empty low-power state with a series of time-indexed idle and low-power empty states as shown in Figure 2.10. The expansion of idle and low-power states into time-indexed states is done only to aid in deriving in the optimal policy. A time-indexed SMDP can contain non-indexed states. Once the policy is obtained, the actual implementation is completely event-driven in contrast to the policies based on discrete-time Markov decision processes. Thus all decisions are made upon event occurrences. The decision to go to a low-power state is made once, upon entry to the idle state as discussed in Section 2.3.2. Other events are user request arrivals or service completions. Note that the technique we present is general, but in this work we will continue to refer to the examples shown in Section 2.2.

If an arrival occurs while in the idle state, the system transitions automatically to the active state. When no arrival occurs during the time spent in a given idle state, the power manager can choose to either stay awake, in which case the system enters the next idle state or to transition into the low-power state. When the transition to the low-power state occurs

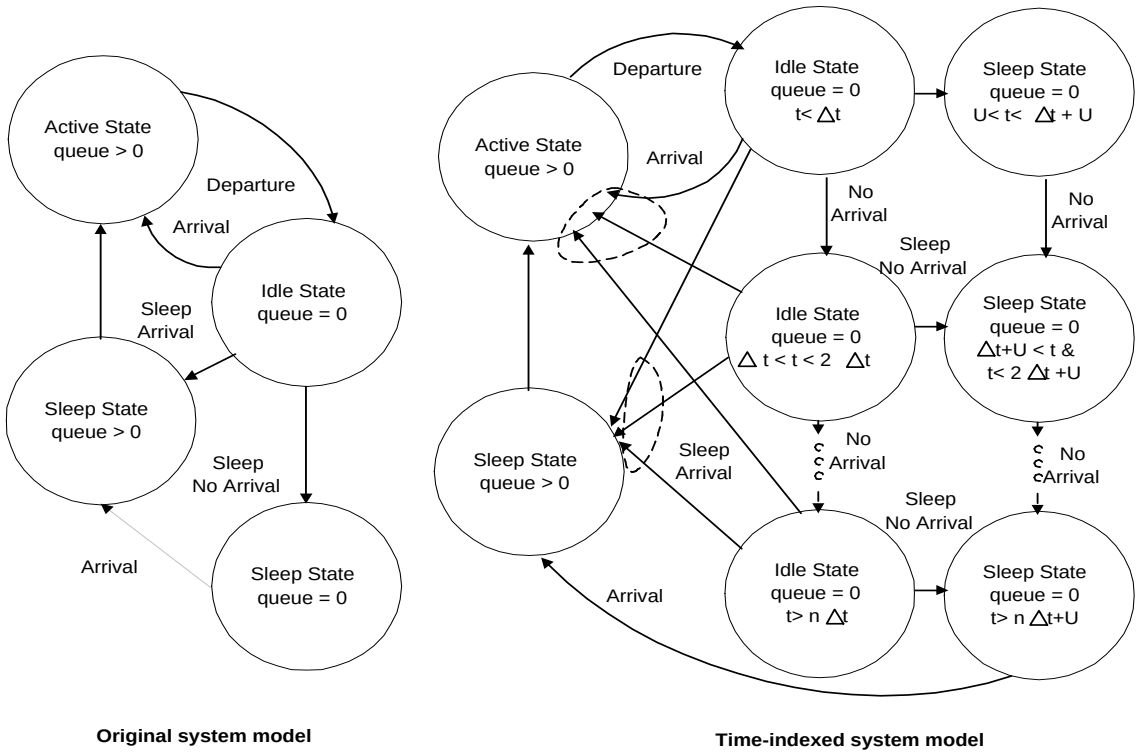

Figure 2.10: Time-indexed SMDP states

from an idle state, the system can arrive to the low-power state with the queue empty or with jobs waiting in the queue. The low-power state with queue empty is indexed by the time from first entry into idle state from active state, much in the same way idle states are indexed, thus allowing accurate modeling of the first arrival. The LP formulation for average-cost SMDP still holds, but the cost, the probability and the expected time functions have to be redefined for time-indexed states in SMDP. Namely, for the time-indexed states Equation 2.12 that calculates cost assigned to the state s_i with action a_i is replaced by:

$$cost(s_i, a_i) = k(s_i, a_i) + \sum_{s_{i+1} \in S_{i+1}} c(s_{i+1}, s_i, a_i) y(s_i, a_i) \tag{2.21}$$

and Equation 2.17 describing the time spent in the state s_i with action a_i is replaced by:

$$y(s_i, a_i) = \int_{t_i}^{t_i+\Delta t} \frac{(1-F(t))dt}{1-F(t_i)} \tag{2.22}$$

The probability of getting an arrival is defined using the time indices for the system state where $t_i \leq t \leq t_i + \Delta t$:

$$p(s_{i+1}|t_i, s_i, a_i) = \frac{F(t_i+\Delta t) - F(t_i)}{1-F(t_i)} \tag{2.23}$$

Equation 2.16 is replaced by the following set of equations. The probability of transition to the next idle state is defined to be $m(s_{i+1}|s_i, a_i) = 1 - p(s_{i+1}|t_i, s_i, a_i)$ and of transition back into the active state is $m(s_{i+1}|s_i, a_i) = p(s_{i+1}|t_i, s_i, a_i)$. The general cumulative distribution of event occurrences is given by $F(t_i)$.

An example below illustrates how the time indexing is done.

Example 2.4.4 *The cumulative distribution of user request arrival occurrences in the idle state follows a Pareto distribution: $F(t_i) = 1 - at_i^{-b}$. The transition from the idle to the low-power state follows uniform distribution with average transition time $t_{ave} = (t_{max_a s} + t_{min_a s})/2$. The time increments are indexed with j. Thus the probability of transition from idle state at time increment $j\Delta t$ into the low-power state with no elements in the queue is given by: $m(s_{i+1}|s_i, a_i) = \frac{1-F(j\Delta t + t_{ave})}{1-F(j\Delta t)}$. This equation calculates the conditional probability that there will be no arrivals up to time $(j+1)\Delta t + t_{ave}$ given that there was no arrival up to time $j\Delta t + t_{ave}$. Note that in this way we are taking history into account. Similarly, we can define the probability of transition from the idle state into a low-power state with an element in the queue by: $m(s_{i+1}|s_i, a_i) = \frac{F(j\Delta t + t_{ave}) - F(j\Delta t)}{1-F(j\Delta t)}$.*

The expected time spent in the idle state indexed with time increment $N\Delta t$ can be defined by: $y(s,a) = \int_{N\Delta t}^{(N+1)\Delta t} \frac{(1-F(t))dt}{1-F(t_i)}$, which after integration simplifies to: $\frac{((N+1)\Delta t)^{1-a} - (N\Delta t)^{1-a}}{(1-a)(N\Delta t)^{-a}}$. With that, we can calculate energy consumed in the idle state, again assuming that there is no fixed energy cost and that the power consumption is defined by P_I: $\frac{P_I((N+1)\Delta t)^{1-a} - (N\Delta t)^{1-a}}{(1-a)(N\Delta t)^{-a}}$.

TISMDP policies are implemented in a similar way to the renewal theory model, but there are more possible decision points. Briefly, upon entry to each decision state, the pseudo-random number *RND* is generated and normalized. The device will transition into low-power state at the time interval for which the probability of going to that state as given by the policy is greater than *RND*. Thus the policy can be viewed as randomized timeout. The device transitions into active state if the request arrives before entry into low-power state. Once the device is turned off, it stays off until the first request arrives, at which point it transitions into active state. The detailed discussion of how the policy is implemented if there is only one decision state has been presented in Section 2.3.2.

Example 2.4.5 *As mentioned in Example 2.4.1,the SmartBadge has two states where decisions can be made: idle and standby. From the idle state, it is possible to give a command to transition to the standby or to the off state. From standby, only a transition to the off state is possible. In this case, both the idle and the standby states are time-indexed. The optimal policy gives a table of probabilities determining when the transition between the idle, the standby and the off states should occur. For example, a policy may specify that if the system has been idle for 50ms, then the transition to the standby state should occur with probability of 0.4, the transition to the off state with probability of 0.2 and otherwise the device would stay idle. Once in the standby state for another 100ms the policy may specify that the transition into the off state should occur with probability of 0.9. When a user request arrives, the system transitions back into the active state.*

In this section, we presented a power management algorithm based on Time-Indexed Semi-Markov Decision Processes. The TISMDP model is more complex than the SMDP model, but is more accurate and is also applicable to a wider set of problems, such as a problem that has more than one non-exponential transition occurring at the same time. The primary difference between the TISMDP model and the renewal theory model is that TISMDP supports multiple decision points in the system model, while renewal theory allows for only one state in which the power manager can decide to transition the device to the low power state. For example, in systems where there are multiple low-power states, the power manager would not only have to make a decision to transition to low-power state, but also could transition the system from one low-power state into another. Renewal theory

cannot be used for this case as there are multiple decision states. The main advantage of the renewal theory model is that it is more concise and thus computes faster. The renewal theory model has only five states, as compared to $O(N)+2$ states in the TISMDP model (N is the maximum time index). In addition, each of the $O(N)$ states require evaluations of one double and two single integrals, compared with a very simple arithmetic formulation for the renewal theory model.

2.5 DPM Results

We perform the policy computation using the solver for linear programs [9] based on the simplex method. The optimization runs in just under 1 minute on a 300MHz Pentium processor. We first verified the optimization results using simulation. Inputs to the simulator are the system description, the expected time horizon (the length of user trace), the number of simulations to be performed and the policy. The system description is characterized by the power consumption in each state, the performance penalty, and the function that defines the transition time probability density function and the probability of transition to other states given a command from the power manager. Note that our simulation used both probability density functions (pdfs) we derived from data and the original traces. When using pdfs, we just verified the correctness of our problem formulation and solution. With real traces we were able to verify that indeed pdfs we derived do in fact match well the data from the real system, and thus give optimal policies for the real systems. The results of the optimization are in close agreement with the simulation results.

In the next sections, we show large savings we **measured** on three different devices: laptop and desktop hard disks and the WLAN card and the simulation results showing savings in power consumption when our policy is implemented in a SmartBadge portable system. As the first three examples (two hard disks and WLAN) have just one state in which the decision to transition to low-power state can be made, the renewal theory model and the TISMDP model give the same results. The last example (SmartBadge) has two possible decision states - idle and standby state. In this case, the TISMDP model is necessary in order to obtain the optimal policy.

2.5.1 Hard Disk

We implemented the power manager as part of a filter driver template discussed in [47]. A filter driver is attached to the vendor-specific device driver. Both drivers reside in the operating system, on the kernel level, above the ACPI driver implementations. Advanced Configuration and Power Interface, ACPI [35], is industry-standard for OS-directed configuration and power management. Application programs such as word processors or spreadsheets send requests to the OS. When any event occurs that concerns the hard disk, power manager is notified. When the PM issues a command, the filter driver creates a power transition call and sends it to the device which implements the power transition using ACPI standard. The change in power state is also detected with the digital multimeter that measures current consumption of the hard disk.

We **measured** and simulated three different policies based on stochastic models and compared them with two bounds: *always-on* and *oracle* policies. Always-on policy leaves the hard disk in the active state, and thus does not save any power. Oracle policy gives the lowest possible power consumption, as it transitions the disk into sleep state with the perfect knowledge of the future. It is computed off-line using a previously collected trace. Obviously, the oracle policy is an abstraction that cannot be used in run-time DPM.

All stochastic policies minimized power consumption under a 10% performance constraint (10% delay penalty). The results are shown in Figures 2.11 and 2.12. These figures best illustrate the problem we observed when user request arrivals are modeled only with the exponential distribution as in the CTMDP model [64]. The simulation results for the exponential model (CTMDP) show large power savings, but measurement results show no power savings and a very high performance penalty. As the exponential model is memoryless, the resulting policy makes a decision as soon as the device becomes idle or after a very short filtering interval (filtered 1s columns in Figures 2.11 and 2.12). If the idle time is very short, the exponential model gets a large performance penalty due to the wakeup time of the device and a considerable cost in shut-down and wakeup energies. In addition, if the decision upon entry to idle state is to stay awake, large idle times, such as lunch breaks, will be missed. If the policy is forced to re-evaluate until it shuts down (exponential), then it will not miss the long idle times. When we use a short timeout to filter out short arrival

times, and force the power manager to re-evaluate its decision (filtered exponential), the results improve. The best results are obtained with our policy. In fact, our policy uses 2.4 times less power than the always-on policy. These results show that it is critically important to not only simulate, but also measure the results of each policy and thus verify the assumptions made in modeling. In fact, we found that modeling based on simple Markov chains is not accurate, and that we do require more complex model presented in this paper.

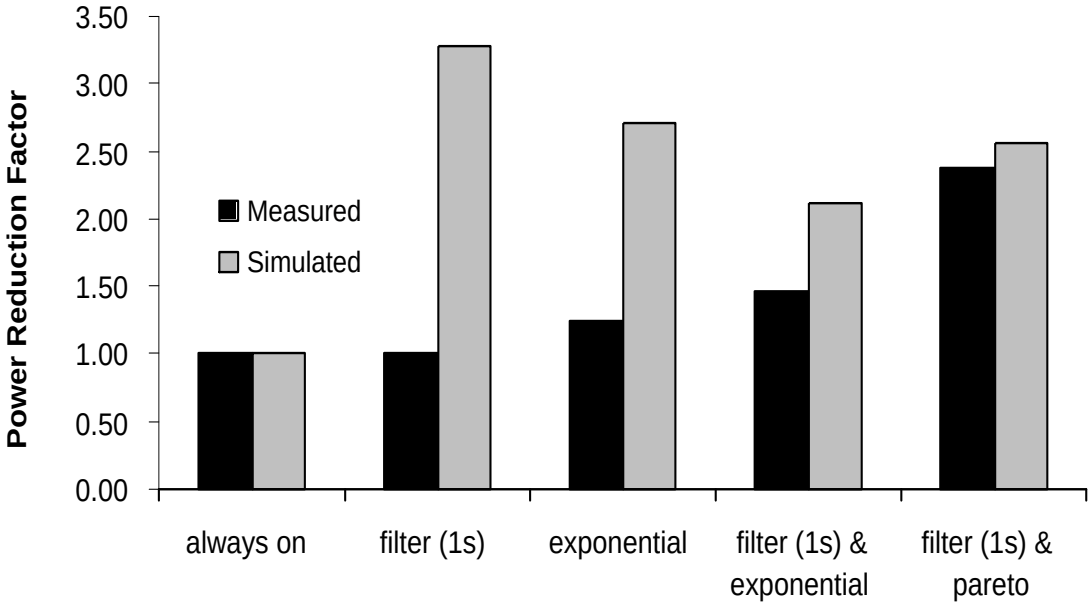

Figure 2.11: Measured and simulated hard disk power consumption

Comparison of all policies measured on the laptop is shown in Table 2.6, and for the desktop in Table 2.7. Karlin's algorithm analysis [43] is guaranteed to yield a policy that consumes at worst twice the minimum amount of power consumed by the policy computed with perfect knowledge of the user behavior. Karlin's policy consumes 10% more power and has worse performance than the policy based on our time-indexed semi-Markov decision process model. In addition, our policy consumes 1.7 times less power than the default Windows timeout policy of 120s and 1.4 times less power than the 30s timeout policy on the laptop. Our policy performs better than the adaptive model [14], and significantly

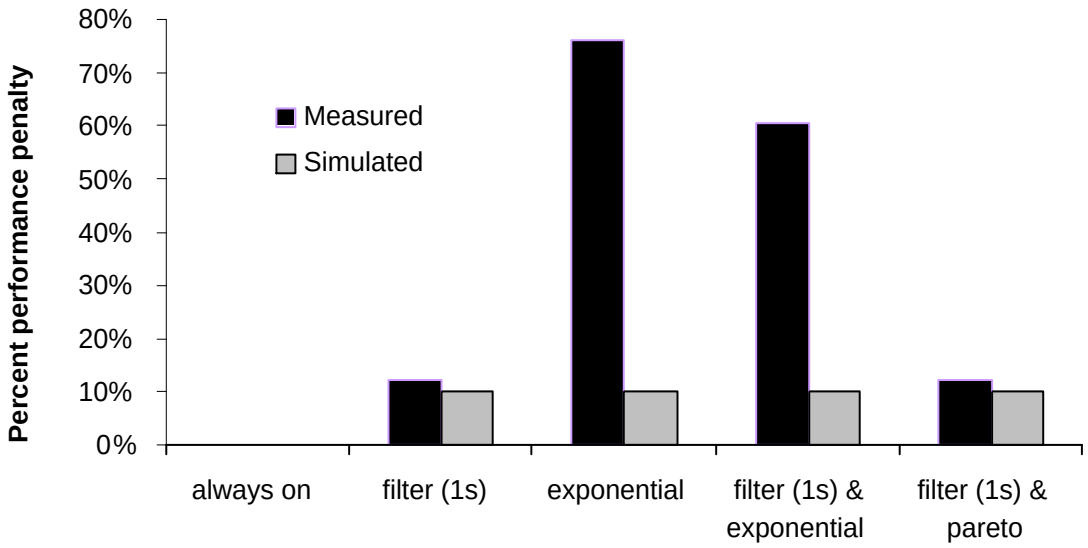

Figure 2.12: Measured and simulated hard disk performance

better than the policy based on discrete-time Markov decision processes (DTMDP). The policy based on the simple continuous-time model (CTMDP) (implemented here without re-evaluation and with initial 1s filter) performs worse then the always-on policy, primarily due to the exponential interarrival request assumption. This policy both misses some long idle periods, and tends to shut-down too aggressively, as can be seen from its very short average sleep time. Similar results can be seen on the desktops. Better overall results can be obtained by using re-evaluations with filtering.

Performance of the algorithms can be compared using three different measures. N_{sd} is defined as the number of times the policy issued sleep command. N_{wd} gives the number of times the sleep command was issued and the hard disk was asleep for shorter than the time needed to recover the cost of spinning down and spinning up the disk. Clearly, it is important to minimize N_{wd} while maximizing N_{sd}. In addition, the average length of time spent in the sleep state (T_{ss}) should be as large as possible while still keeping the power consumption down. From our experience with the user interaction with the hard disk, our algorithm performs well, thus giving us low-power consumption with still good

Table 2.6: Laptop Hard Disk Measurement Comparison

Algorithm	Pwr (W)	N_{sd}	N_{wd}	$T_{ss}(s)$
Oracle	0.33	250		118
Ours	0.40	326	76	81
Adaptive	0.43	191	28	127
Karlin's	0.44	323	64	79
30s timeout	0.51	147	18	142
DTMDP	0.62	173	54	102
120s timeout	0.67	55	3	238
always on	0.95	0	0	0
CTMDP	0.97	391	359	4

Table 2.7: Desktop Hard Disk Measurement Comparison

Algorithm	Pwr (W)	N_{sd}	N_{wd}	$T_{ss}(s)$
Oracle	1.64	164	0	166
Ours	1.92	156	25	147
Karlin's	1.94	160	15	142
Adaptive	1.97	168	26	134
30s timeout	2.05	147	18	142
120s timeout	2.52	55	3	238
DTMDP	2.60	105	39	130
always on	3.48	0	0	0
CTMDP	3.90	326	318	4

performance.

As mentioned earlier, we filtered request arrivals using a fraction of hard disk break-even time. The effect of filtering arrivals into the idle state is best shown in Figure 2.13 for the policy with the performance penalty of the laptop hard disk limited to 10%. For very short filter times the power consumption is very high since the overhead of transition to and from the low-power state has not been compensated. The power consumption grows for longer filter times since more time is spent in the idle state before transitioning to the low-power state, thus wasting some power. Note that the best filtering intervals are on the order of seconds since the hard disk break-even time is also on the order of seconds.

The event-driven nature of our algorithm, as compared to algorithms based on discrete time intervals, saves considerable amount of power while in sleep state as it does not require

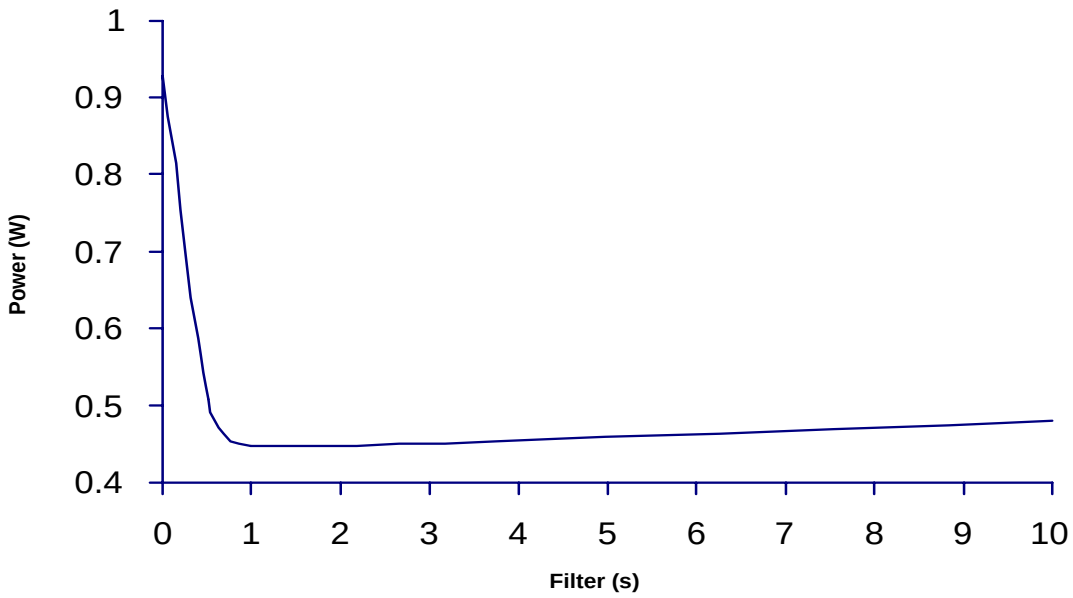

Figure 2.13: Power consumption vs. filter size

policy evaluation until an event occurs. Waking up a 10 W processor every 1s for policy re-evaluation that takes 100ms to execute would use 1800 J of energy during a normal 30 minute break. With an event-driven policy, the processor could be placed in a low-power mode during the break time, thus saving a large portion of battery capacity.

2.5.2 WLAN card

For WLAN measurements we used Lucent's WLAN 2Mb/s card [50] running on the laptop. As a mobile environment is continually changing, it is not possible to reliably repeat the same experiment. As a result, we needed to use a trace-based methodology discussed in [56]. The methodology consists of three phases: collection, distillation and modulation. We used *tcpdump* [40] utility to get the user's trace for two different applications: web browsing and telnet. During distillation we prepared the trace for the modeling step. We had a LAN-attached host read the distilled trace and delay or drop packets according to the parameters we obtained from the measurements. In this way, we were able to recreate the

experimental environment, so that different algorithms can be reliably compared.

We implemented three different versions of our algorithm for each application, each with different power (P_{ave}) and performance penalty ($T_{penalty}$). The algorithms are labeled *Ours a,b,c* for web browser and *Ours 1,2,3* for telnet. Since web and telnet arrivals behave differently (see Figure 2.4), we observe through the OS what application is currently actively sending and use the appropriate power management policy. The performance penalty for WLAN is a measure of the total overhead time due to turning off the card. Note that for the hard disk we measured instead the average time in the sleep state, as the accurate real overhead was difficult to obtain. In addition to measuring the energy consumption

Table 2.8: DPM for WLAN Web Browser

Policy	N_{sd}	N_{wd}	$T_{penalty}$ (sec)	P_{ave} (W)
Oracle	395	0	0	0.467
Ours (a)	363	96	6.90	0.474
Ours(b)	267	14	1.43	0.477
Karlin's	623	296	23.8	0.479
Ours(c)	219	9	0.80	0.485
CTMDP	3424	2866	253.7	0.539
Default	0	0	0	1.410

(and then calculating average power), we also quantified the performance penalty using three different measures. Delay penalty, T_p, is the time the system had to wait to service a request since the card was in the sleep state when it should not have been. In addition, we measure the number of shutdowns, N_{sd} and the number of wrong shutdowns, N_{wd}. A shutdown is viewed as wrong when the sleep time is not long enough to make up for the energy lost during transition between the idle and off state. The number of shutdowns is a measure of how eager the policy is, while a number of wrong shutdowns tells us how accurate the policy is in predicting a good time to shut down the card.

The measurement results for a 2.5hr web browsing trace are shown in Table 2.8. Our algorithms (*Ours a,b,c*) show, on average, a factor of three in power savings with a low performance penalty. Karlin's algorithm [43] is guaranteed to be within a factor of two of the oracle policy. Although its power consumption is low, it has a performance penalty that

Table 2.9: DPM for WLAN Telnet Application

Policy	N_{sd}	N_{wd}	$T_{penalty}$ (sec)	P_{ave} (W)
Oracle	766	0	0	0.220
Ours(1)	798	21	2.75	0.269
Ours(2)	782	33	2.91	0.296
Karlin's	780	40	3.81	0.302
Ours(3)	778	38	3.80	0.310
CTMDP	943	233	20.53	0.361
Default	0	0	0	1.410

is an order of magnitude larger than for our policy. A policy that assumes the exponential arrivals only, CTMDP [64], has a very large performance penalty because it makes the decision as soon as the system enters idle state.

Table 2.9 shows the measurement results for a 2hr telnet trace. Again our policy performs best, with a factor of five in power savings and a small performance penalty. The telnet application allows larger power savings because on average it transmits and receives much less data then the web browser, thus giving us more chances to shut down the card.

2.5.3 SmartBadge

In contrast to the previous examples, where we implement and measure the decrease in power consumption when using our power management policies, in this case we perform a case study on the tradeoffs between power and performance for the SmartBadge. The SmartBadge is a good example of a system that has more than one decision point and thus requires the TISMDP model in order to obtain an optimal policy. We first study the tradeoffs between power and performance for policies with just one decision state (idle state), and then follow with an example contrasting policies with one state to policies that have two decision states (idle and standby).

The results of simulation shown in Figure 2.14 clearly illustrate the tradeoff between different policies for one decision state that can be implemented in the SmartBadge system. The performance penalty is defined as the percent of the time system spends in a low-power state with a non-empty queue. In general, the goal is to have as few requests as possible

waiting for service. For systems with a hard real-time constraint, this penalty can be set to large values to force less aggressive power management, thus resulting in less requests queued up for service. In systems where it is not as critical to meet time deadlines, the system can stay in a low-power state longer, thus accumulating more requests that can be serviced upon return to the active state.

Because of the particular design characteristics of the SmartBadge, the tradeoff curves of performance penalty and power savings are very close to linear. When the probability of going to sleep is zero, no power can be saved, but the performance penalty can be reduced by 85% as compared to the case where the probability is one. On the other hand, about 50% of the power can be saved when the system goes to standby upon entry to idle state.

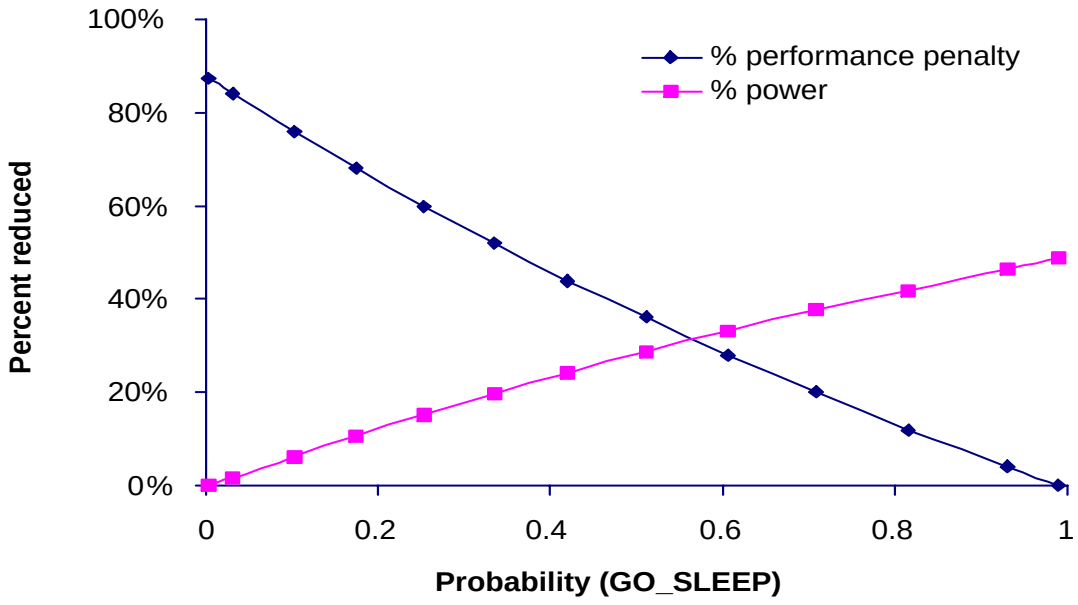

Figure 2.14: SmartBadge DPM results

In addition to analyzing power and performance tradeoffs for policies that have only one decision state, we have also compared the one decision state (idle) policy to a policy with two decision states (idle and standby) with the same performance penalty. The results in Table 2.10 clearly show that considerably larger power savings with the same performance penalty can be obtained when using a more complex policy optimization model

that enables multiple decision points (TISMDP model) instead of just one decision point (renewal theory model).

Table 2.10: Comparison of Policies by Decision State Number

No. Decision States	Power (W)
One state	1.84
Two states	1.47

2.6 Summary

Dynamic power management policies reduce energy consumption by selectively placing components into low-power states. In contrast to heuristic policies, such as timeouts, policies based on stochastic models can guarantee optimal results. The quality of results of stochastic DPM policies depends strongly on the assumptions made. In this chapter I present and implement two different stochastic models for dynamic power management. The measurement results show large power savings.

The first approach requires that only one decision point be present in the system. This model is based on renewal theory. The second approach allows for multiple decision points and is based on Semi-Markov Decision Process (SMDP) model. The basic SMDP model can accurately model only one non-exponential transition occurring with the exponential ones. I presented TISMDP model as the extension to SMDP model in order to describe more than one non-exponential transition occurring at the same time. TISMDP model is very general, but also is more complex. Thus, it should be used for very general systems that have more than one decision point (and thus multiple states that trade-off power for performance). One limitation of both techniques is that they assume a prior knowledge of the stochastic characteristics of the system and the workload. Since the workload is often non-stationary, it is important to adapt to the changes. The adaptation can be realized with either model using an approach presented in [14]. Note that I have measured large savings in power using my approaches although the traces used for measurements consisted of

multiple users and applications.

Large power savings have been observed when using my approach on four different portable devices: the laptop and the desktop hard disks, the WLAN card and the SmartBadge. The measurements for the hard disks show that my policy gives as much as 2.4 times lower power consumption as compared to the default Windows timeout policy. In addition, my policy obtains up to 5 times lower power consumption for the wireless card relative to the default policy. The power management results on the SmartBadge show savings of as much as 70% in power consumption. Finally, the comparison of policies obtained for the SmartBadge with the renewal model and TISMDP model clearly illustrate that whenever there is more than one decision point available, the TISMDP model should be used as it can utilize the extra degrees of freedom and thus obtain an optimal power management policy.

Chapter 3

Dynamic Voltage Scaling

3.1 Introduction

Dynamic Voltage Scaling (DVS) algorithms adjust the device speed and voltage according to the workload at run-time. Since most systems do not need peak performance at all times, decreasing the device speed and voltage during less busy periods increases energy efficiency. Another approach is to only reduce clock frequency. Although this does decrease the power consumption, it does not significantly alter the energy consumption because the energy consumption is directly proportional to the execution time. It is also possible to decrease voltage in addition to lowering the clock frequency. Lower voltage also causes a decrease in the peak performance. Thus, the best savings can be obtained with DVS algorithms that dynamically adjust both voltage and frequency to computational load. Implementing a DVS algorithm for a processor requires both hardware and software support that is not commonly available yet, even though there have been a few examples of DVS implementation such as in [24].

In this chapter I extend the DPM model discussed in the previous chapter with a DVS algorithm, thus enabling larger energy savings. The DVS algorithm assumes the same stochastic system model that was assumed in previous chapter: a set of power states whose transitions are described with stochastic distributions. In contrast to the DPM model, the DVS model expands the active state into a set of states characterized by different energy and performance trade-offs. In addition, the DVS model handles workload non-stationarity.

The DVS algorithm is implemented for the SmartBadge portable device presented in Chapters 1 and 2. All SmartBadge components have four main power states: *active, idle, standby* and *off*. In addition, the processor can operate over a range of frequencies. For each frequency, there is a minimum allowed voltage of operation. If the processor runs at the minimum frequency and voltage required to sustain the performance level required by the application, it is possible to save power even when the system is active, in addition to the savings that can be obtained by DPM during idle periods. This principle is exploited by the recently announced Transmeta's Crusoe processor [24].

The first contribution of this work is to develop and verify a stochastic model for prediction of execution times for streaming multimedia applications on a frame-by-frame basis. My model is based on the change-point detection theory used for ATM traffic detection among other applications [82]. I compare my model to ideal prediction (requires knowledge of future) and to the exponential moving average used in [62]. The prediction algorithm developed is then used as a part of a power control strategy that merges DVS and DPM.

The second contribution of this work is to merge the DPM and the DVS approaches, by expanding the active state definition to include multiple settings of frequency and voltage, thus resulting in a range of performance and power consumptions available for tradeoff at run time. In this way, the power manager can control performance and power consumption levels both by using DVS when the system is active, and by transitioning components into low-power states when the system is idle.

Section 3.2 describes the stochastic models of the system components. The models are based on experimental observations. In Section 3.3 I present the theoretical basis for detection of rate change together with dynamic selection of CPU frequency and voltage. I show simulation and measurement results for MPEG2 CIF size video and MP3 audio running on the SmartBadge in Section 3.4.

3.2 System Model

The system can be modeled with three components: the user (a source of external events), the device (i.e. SmartBadge) and the queue (the buffer associated with the device) as shown

in Figure 2.1. The power manager observes all event occurrences of interest and takes decisions on what state the system should transition to next, in order to minimize energy consumption for a given performance constraint. While the device is active, the power manager selects the most appropriate execution frequency and voltage for the processor. As our work was motivated by a real design of the SmartBadge, in all our examples we use the SmartBadge hardware with MPEG2 video (CIF size) and MP3 audio.

Each system component is described probabilistically. The user behavior is modeled by a *request interarrival distribution*. For streaming multimedia applications, requests represent frame arrivals from the network. Similarly, the *service time distribution* describes the behavior of the device in the active state. In multimedia case, it represents the time needed for processing a frame (decompressing it and sending to the output interface). The *transition time distribution* models the time taken by the device to transition between its power states. Finally, the combination of interarrival time distribution (incoming frame arrivals) and service time distribution (frame decoding times) characterizes well the behavior of the queue (frame buffer). The details of each system component are described in the next sections.

3.2.1 Portable Device

Portable devices typically have multiple power states. Each device has one active state in which it services user requests, and one or more inactive low-power states as shown in Figure 3.1. The active state can further be characterized by a set of sub-states differentiated by performance (e.g. CPU frequency) and power consumption (e.g. CPU voltage). The power manager can trade off power for performance by placing the device into low-power states or by scaling CPU frequency and voltage. Each low power state can be characterized by the power consumption and the performance penalty incurred during the transition to or from that state. Usually higher performance penalty corresponds to lower power states.

The SmartBadge Device

The SmartBadge embedded system is shown in Figure 1.2. Components in the SmartBadge, the power states and the transition times of each component from standby (t_{sby}) and

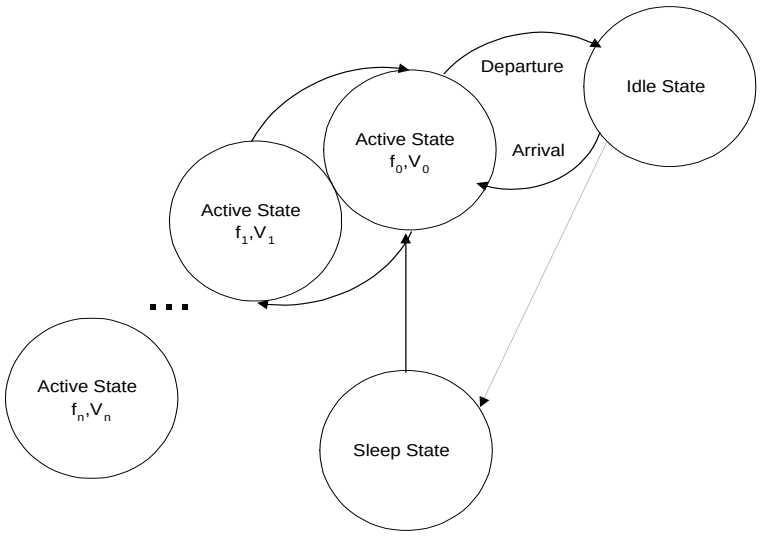

Figure 3.1: A set of active and low-power states

off (t_{off}) state into active state are shown in Table 2.1.

The StrongARM processor on the SmartBadge can be configured at run-time by a simple write to a hardware register to execute at one of 10 different frequencies. Note that the number of frequencies is predefined by the design of the StrongARM processor. We measured the transition time between two different frequency settings at 150 microseconds. Since typical decoding time for MPEG video or MP3 audio is much longer than the transition time, it is possible to change the CPU frequency without perceivable overhead. For each frequency, there is a minimum voltage the SA-1100 needs in order to run correctly, but with lower energy consumption. Figure 3.2 shows the frequency-voltage tradeoff.

In addition to the active state, the SmartBadge supports three lower power states: *idle, standby and off*. The idle state is entered immediately by each component in the system as soon as that particular component is not accessed. The standby and off state transitions can be controlled by the power manager. The transition from standby or off state into the active state can be best described using the uniform probability distribution.

The Active State Model

Service times (decoding times for video or audio frames) on the SmartBadge in the active state are modeled by an exponential distribution. The average service time is defined by

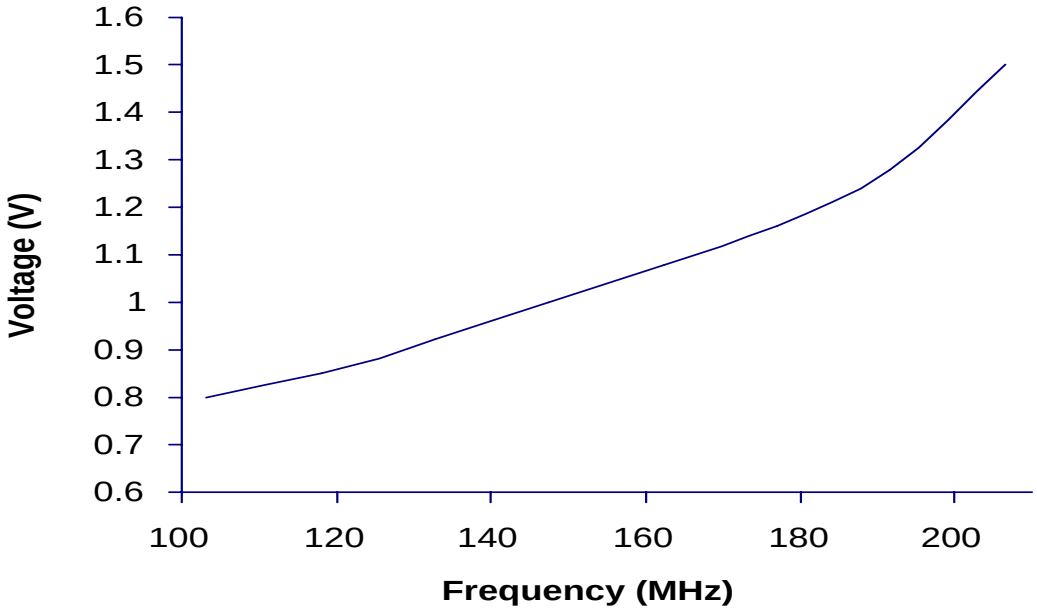

Figure 3.2: Frequency vs. Voltage for SA-1100

$\frac{1}{\lambda_D}$ where λ_D is the average service rate (measured in frames/second for MPEG video and MP3 audio) as discussed in Chapter 2. Equation 3.1 defines the cumulative probability of the device servicing a user request within time interval t.

$$F_D(t) = 1 - e^{-\lambda_D t} \tag{3.1}$$

Figure 3.3 shows the tradeoff between performance and energy when running MP3 audio decode on the SmartBadge hardware at allowable frequency and voltage setting for the SA-1100 processor, and Figure 3.4 shows the same results for MPEG video. The shape of the performance curve versus processor frequency setting depends on the application and on the underlying hardware. MP3 audio was decoded using slower SRAM on the SmartBadge. Since memory access time does not depend on processor clock frequency, performance improvements at high processor frequencies are memory-bound, and speedup is not linear. MPEG video decode ran on much faster SDRAM and thus its performance curve is almost linear as it is more limited by the processor speed. In both figures all values

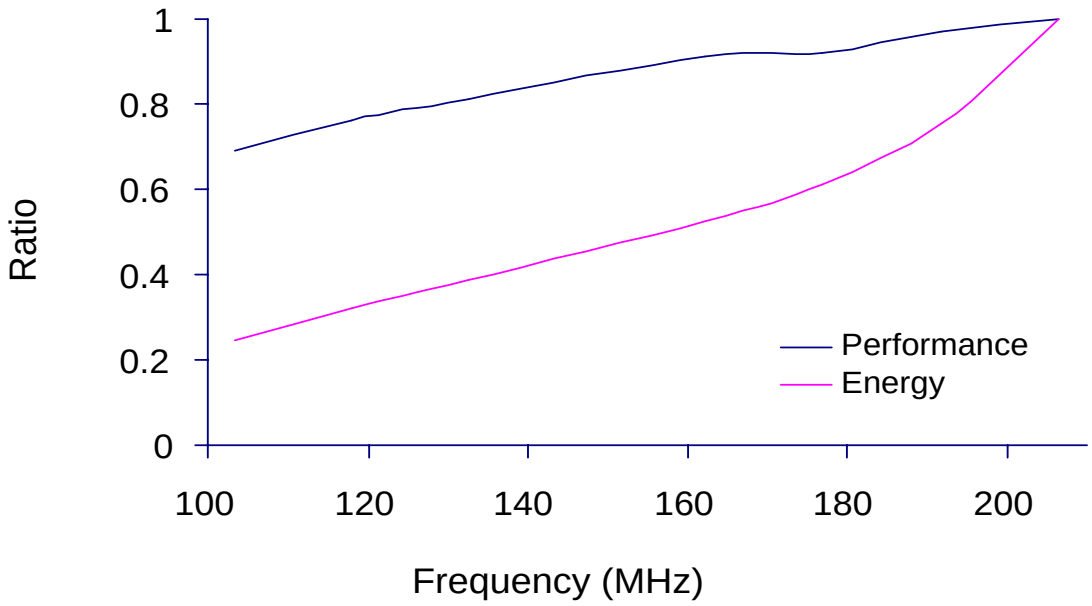

Figure 3.3: Performance and energy for MP3 audio

are normalized to the data points obtained for the fastest frequency.

The basic rationale for DVS is that for frames that take a shorter time to decode, processor frequency and voltage can be lowered, and for longer frames, increased. In addition, the decoding speed needs to be adjusted to frame arrival frequency, so that the frame buffer does not contain too many or too few frames. The detection of changes in decoding speed and arrival frequency are thus critical for optimal setting of CPU frequency and voltage. We present an optimal way for detection in Section 3.3.

3.2.2 User Model

User is a source of external events to the device. The requests to the multimedia application during the decoding are in form of audio or video frame arrivals through the WLAN. Thus, the user's stochastic model in the active state can be defined by the frame interarrival time distribution. We measured MPEG2 video (CIF size) and MP3 audio frame arrival times by monitoring the accesses to the WLAN card. Since the frames arrive through the wireless

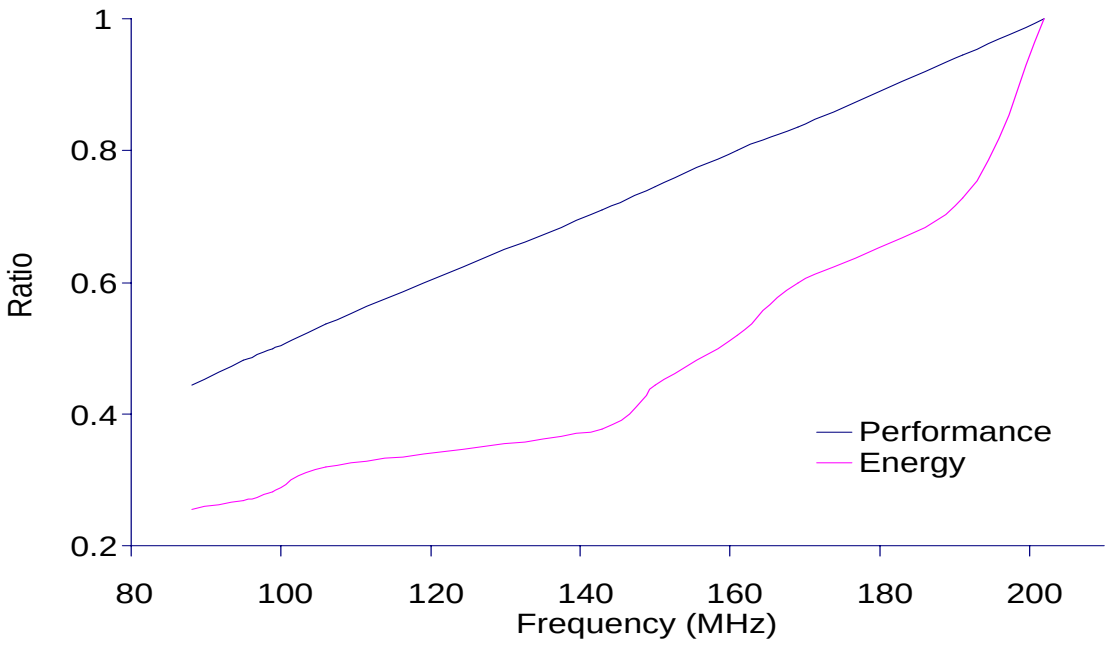

Figure 3.4: Performance and energy for MPEG video

network, the arrival times are not fixed. The frame interarrival times in the active state for both applications can be approximated with an exponential distributions. Figure 3.5 shows the exponential cumulative distribution fitted to the measured results for the MPEG video. Similar results have been observed for the MP3 audio. Frame arrival rate in the active state is defined as λ_U and the mean frame interarrival time is $\frac{1}{\lambda_U}$. The probability of the SmartBadge receiving a frame within time interval t follows the cumulative probability distribution shown below.

$$F_U(t) = 1 - e^{-\lambda_U t} \tag{3.2}$$

Note that the exponential distribution is *not* used to model the arrivals in the idle state, the same way as in the DPM model discussed in Chapter 2. In the idle state, audio or video frames have all been decoded and no new requests have arrived yet from the user. This is when the power manager can make a decision on what low-power state to place the device in as discussed in [73]. The full optimization model should not only decide when

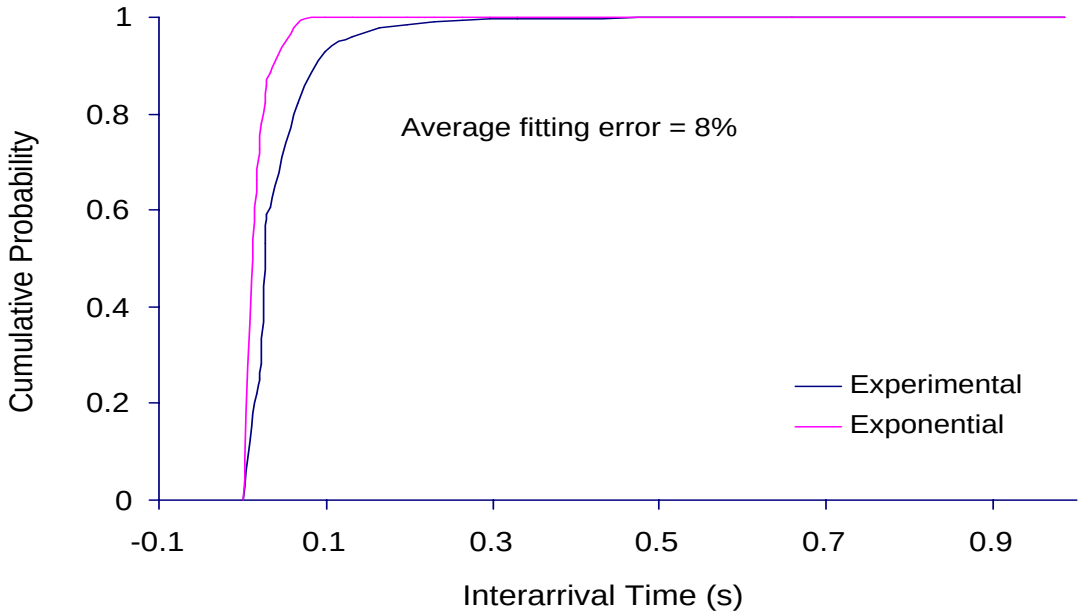

Figure 3.5: MPEG video arrival time distribution

to transition the device into one of the low-power states (standby or off) but should also perform dynamic voltage scaling in the active state.

3.2.3 Queue

Portable devices normally have a buffer for storing requests that have not been serviced yet. For multimedia requests such as MPEG video and audio it is convenient to describe queue in terms of the number of frames waiting in the frame buffer. As the frames arriving to the SmartBadge do not have priority, our queue model contains only the number of frames waiting service (decoding). In the active state, where the exponential distributions is used to describe frame arrivals and service times, the behavior of the system can be modeled using M/M/1 queue model. More details on this model and its application to dynamic voltage scaling are given in the following section.

3.3 Theoretical Background

In Chapter 2 the power manager's only job is to decide when to transition the device into one of the low-power states. Power management policies are obtained using one of two models: renewal theory model and time-indexed semi-markov process model. It was observed that in the idle state we need to accurately model the tail of the interarrival time distribution, which does not follow a perfect exponential distribution. As a result, the time elapsed since the last entry into the idle state had to be accounted for in the model in order to obtain the optimal power management policy. Renewal theory naturally accounts for the time elapsed in the idle state through formulation of the system renewal time. In the TISMDP model, instead of the simple state model shown on the left in Figure 3.6, it was necessary to expand the idle and the sleep states with time index representing elapsed time since the last entry into the idle state as shown on the right. Note that in both renewal and TISMDP models there is only one active state (with one or more elements in the queue).

In this work we have extended the function of power manager (PM) to include decisions on the CPU frequency and voltage setting while in the active state. Thus, instead of having only one active state as shown in Figure 3.6, now there is a set of active states, each characterized by different performance (CPU frequency) and power consumption (CPU voltage) as shown in Figure 3.7. Since TISMDP and renewal models both assumed that active state can be described using the exponential distribution, the transformation from one active into multiple active states is completely compatible with the rest of the model. As a result, the power management policies we develop can make decisions for both dynamic voltage setting and the transition into the low-power states.

At run-time, the PM observes user request arrivals and service completion times (in our case frame arrivals and decoding times), the number of jobs in the queue (the number of frames in the buffer) and the time elapsed since last entry into idle state. When in the active state, the PM checks if the rate of incoming or decoding frames has changed, and then adjusts the CPU frequency and voltage accordingly. Once the decoding is completed, the system enters idle state. At this point the power manager observes the time spent in the idle state, and depending on the policy obtained using either renewal theory or TISMDP model, it decides when to transition into one of the sleep states. When a request from the

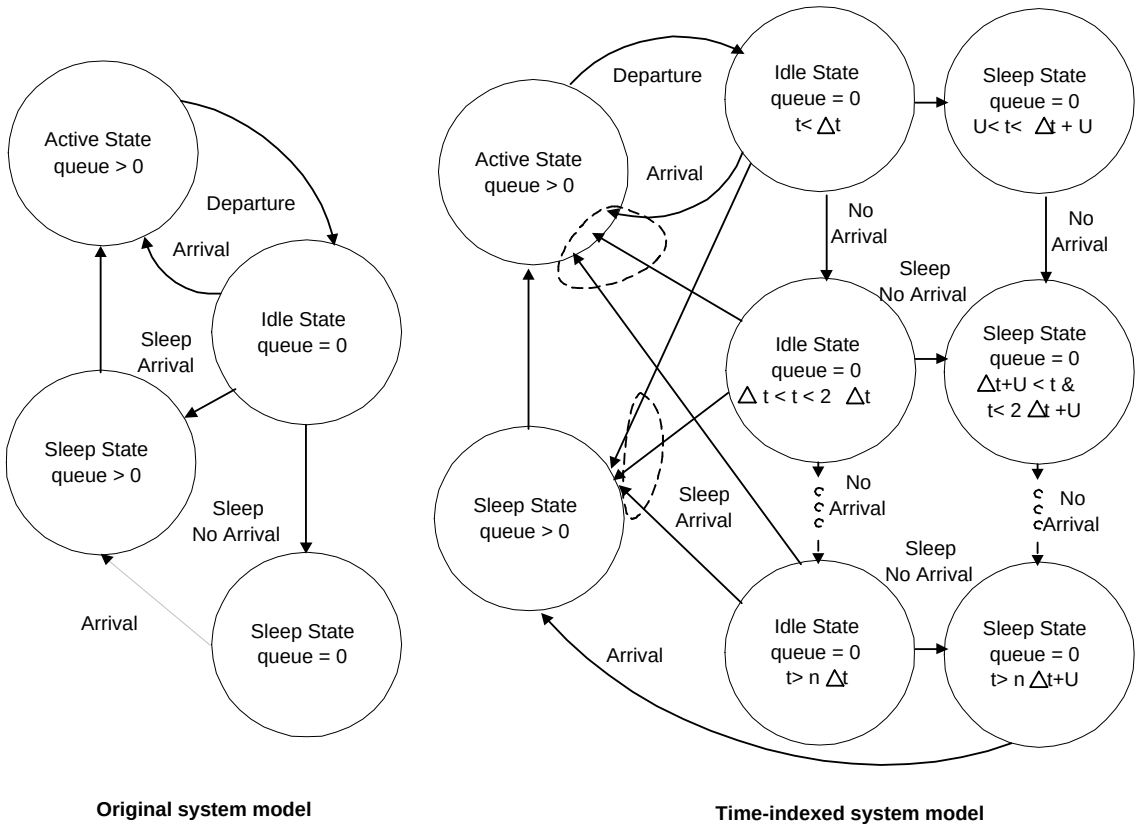

Figure 3.6: Time-indexed SMDP states

user arrives for more audio or video decoding, the power manager transitions the system back into the active state and starts the decoding process.

We next present the optimal approach for detecting a change in the frame arrival or decoding times. Once a change is detected, a decision has to be made on how to set the CPU frequency and voltage. We present results based on M/M/1 queue theory that enable power manager to make this decision.

3.3.1 Dynamic Voltage Scaling Algorithm

The DVS algorithm consists of two main portions: detection of the change in request arrival or servicing rate, and the policy that adjusts the CPU frequency and voltage. The detection

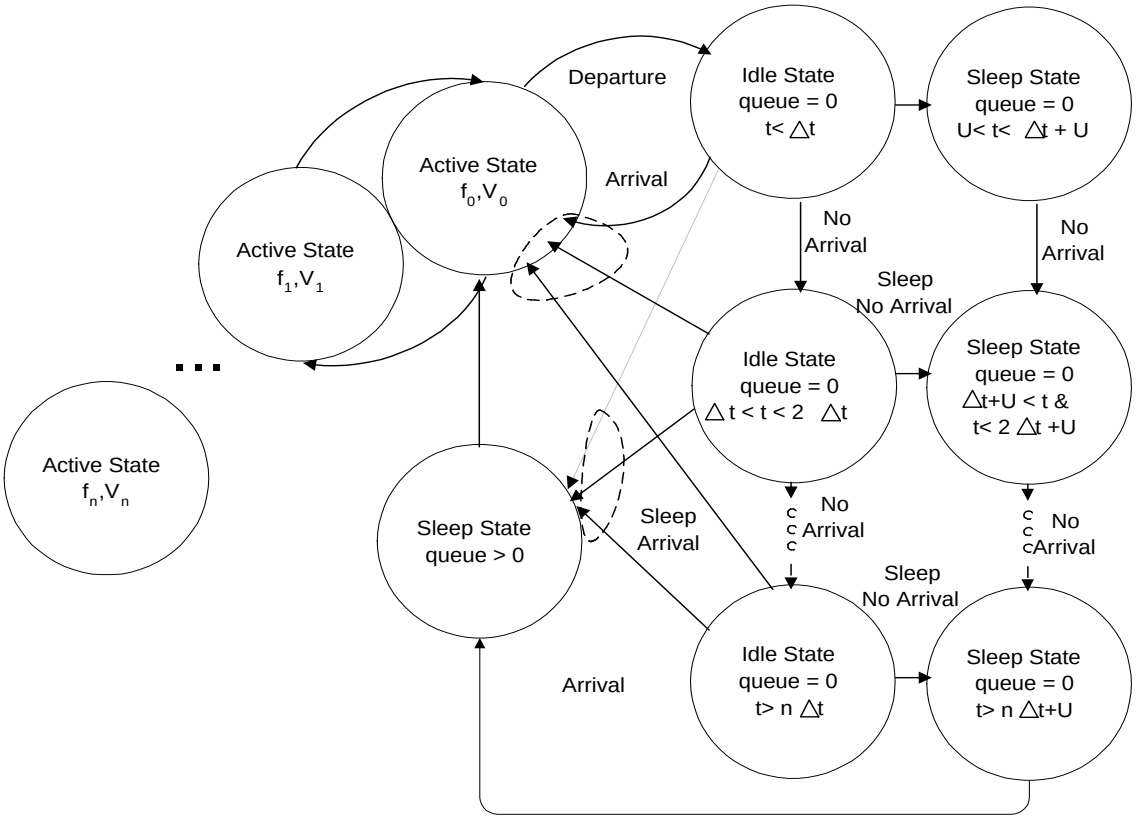

Figure 3.7: Expansion of the active state

is done using maximum likelihood ratio that guarantees optimal detection for the exponential distributions. Policy is implemented using M/M/1 queue results to ensure constant average delay experienced by buffered frames.

Detecting the change in rate is a critical part of optimally matching CPU frequency and voltage to the requirements of the user. For example, the rate of MP3 audio frames coming via RF link can change drastically due to changes in the environment. The servicing rate can change due to variance in computation needed between MPEG frames [5, 13], or just by changing the audio source currently decoded by the MP3 audio. The request (frame) interarrival times and servicing (decoding) times follow the exponential distribution as discussed in the previous section. The two distributions are characterized by the arrival rate, λ_U, and the servicing rate, λ_D.

The change point detection is performed using maximum likelihood ratio, P_{max}, as

shown in Equation 3.3. Maximum likelihood ratio computes the ratio between the probability that a change in rate did occur (numerator in Equation 3.3) and the probability that rate did not change (denominator). The probability that the rate changed is computed by fitting the exponential distribution with an old rate, λ_o, to the first $k-1$ interarrival or decoding times (x_j), and another exponential distribution with a new rate, λ_n, to the rest of the points observed in window of size m (which contains the last m interarrival times of user requests). The probability that the rate did not change is computed by fitting the interarrival or decoding times with the exponential distribution characterized by the current (or old) rate, λ_o.

$$P_{max} = \frac{\Pi_{j=1}^{k-1}\lambda_o e^{-\lambda_o x_j}\Pi_{j=k}^{m}\lambda_n e^{-\lambda_n x_j}}{\Pi_{j=1}^{m}\lambda_o e^{-\lambda_o x_j}} \tag{3.3}$$

An efficient way to compute the maximum likelihood ratio, P_{max}, is to calculate the natural log of P_{max} as shown below:

$$ln(P_{max}) = (m-k+1)ln\frac{\lambda_n}{\lambda_o} - (\lambda_n - \lambda_o)\sum_{j=k}^{m} x_j \tag{3.4}$$

Note that in this equation, only the sum of interarrival (or decoding) times needs to be updated upon every arrival (or service completion). A set of possible rates, Λ, where $\lambda_n, \lambda_o \in \Lambda$ is predefined, as well as the size of the window m. Variable k is used to locate the point in time when the rate has changed. The change point detection algorithm consists of two major tasks: off-line characterization and on-line treshold detection.

Off-line characterization is done using stochastic simulation of a set of possible rates to obtain the value of $ln(P_{max})$ that is sufficient to detect the change in rate. The results are accumulated in a histogram, and then the value of maximum likelihood ratio that gives very high probability that the rate has changed is chosen for every pair of rates under consideration. In our work we selected 99.5% likelihood.

On-line detection collects the interarrival time sums at run time and calculates the maximum likelihood ratio. If the maximum likelihood ratio computed is greater than the one obtained from the histogram, then there is 99.5% likelihood that the rate change occurred, and thus the CPU frequency and voltage need to be adjusted. We found that a window of

$m = 100$ is large enough. Larger windows will cause longer execution times, while much shorter windows do not contain statistically large enough sample and thus give unstable results. In addition, the change point can be checked every $k = 10$ points. Larger values of k interval mean that the changed rate will be detected later, while with very small values the detection is quicker, but also causes extra computation. The same change point detection algorithm can be used for any type of distribution, not only for the exponential distributions.

The adjustment of frequency and voltage is done using M/M/1 queue model [77, 67]. Using this model we try to keep the average total delay for processing frames in the queue constant (Equation 3.5). Note that when general distributions are used, M/M/1 queue model is not applicable, so another method of frequency and voltage adjustment is needed.

$$Frame_{delay} = \frac{\lambda_D}{\lambda_U(\lambda_U - \lambda_D)} \tag{3.5}$$

When either interarrival rate, λ_U, or the servicing rate, λ_D, change, the frame delay is evaluated and the new frequency and voltage are selected that will keep the frame delay constant. For example, if the arrival rate for MP3 audio changes, Equation 3.5 is used to obtain required decoding rate in order to keep the frame delay (and thus performance) constant. The decoding rate can be related back to the processor frequency setting using Figure 3.3 or an equivalent table. On the other hand, if a different frame decoding rate is detected while processor is set to the same frequency, then piece-wise linear approximation based on the application frequency-performance tradeoff curve (Figures 3.3 and 3.4) is used to obtain the new processor frequency setting. In either case, when CPU frequency is set to a new value, the CPU voltage is always adjusted according to Figure 3.2.

3.4 DVS Results

We implemented the change point detection algorithm as a part of the power manager for both MPEG2 video (CIF size) and MP3 audio examples. During the times that the system is idle, the DPM algorithms described in Chapter 2 decide when to transition the system into a low-power state. When the system is in the active state (the state where audio and

video decoding occur), the power manager (PM) observes changes in the frame arrival and decoding rates using change point detection algorithm described in the previous section. Once a change is detected, the PM evaluates the required value of the processor frequency that would enable the frame delay expressed in Equation 3.5 to remain constant. The CPU voltage is set using results shown in Figure 3.2. Figure 3.8 shows the relationship between CPU frequency and MPEG video frame arrival and decoding rates for average buffered frame delay of 0.1 seconds, which then corresponds to an average of 2 extra frames of video buffered. This example is for a clip of football video decoded on the SmartBadge. Similar results can be obtained for other clips, but with different decoding rates, as the rates depend on the content and on the hardware architecture.

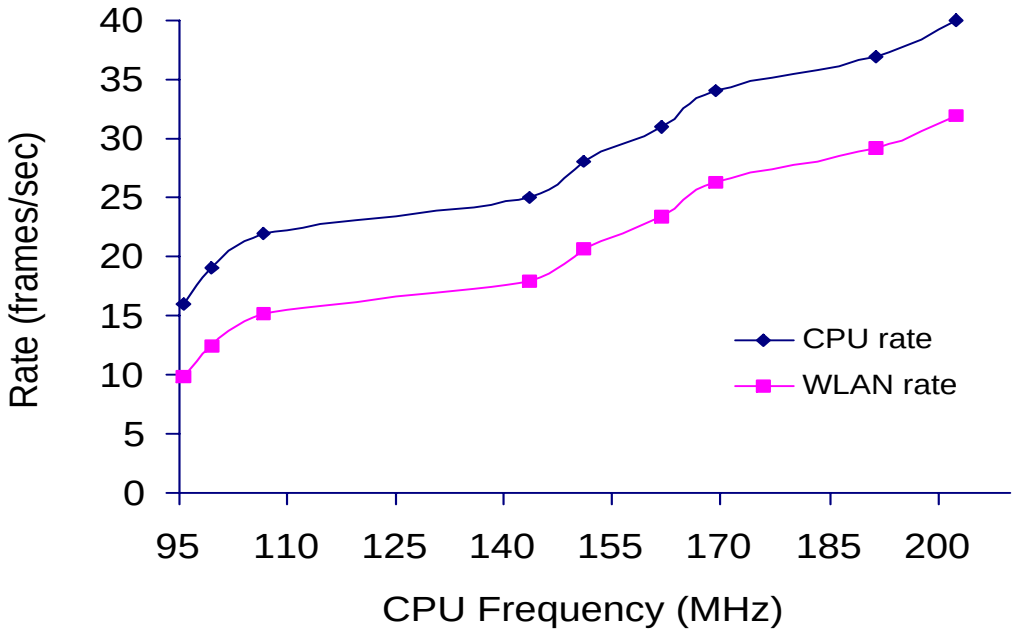

Figure 3.8: MPEG Frame Rates vs. CPU Frequency

We compare our rate change detection algorithm to ideal detection and to the exponential moving average algorithm. Ideal detection assumes knowledge of the future; thus the system detects the change in rate exactly when the change occurs. The exponential moving

average can be defined as follows:

$$Rate_{ave}^{new} = (1-g)Rate_{ave}^{old} + gRate^{cur} \tag{3.6}$$

where $Rate_{ave}^{new}$ is the new average rate, $Rate_{ave}^{old}$ is the old average, $Rate^{cur}$ is the current measured rate and g is the gain. Figure 3.9 shows the comparison results for detecting a change from 10 fr/sec to 60 fr/sec. Our algorithm detects the correct rate within 10 frames and is more stable than either of the two the exponential moving average algorithms (they differ in the value of gain).

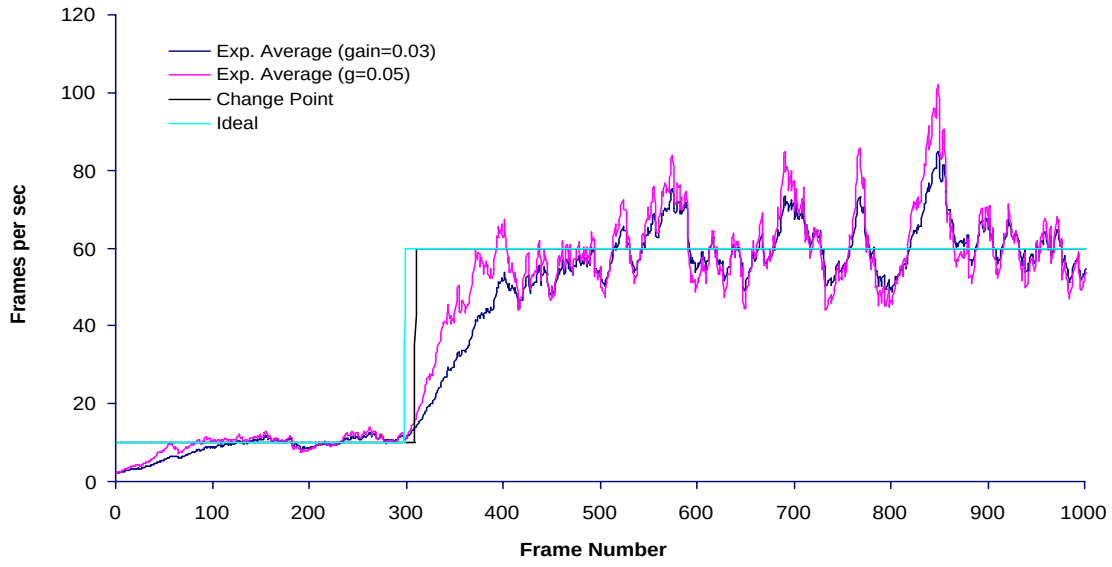

Figure 3.9: Rate Change Detection Algorithms

In the following set of results we compare (i) the ideal detection algorithm, (ii) the exponential average approximation used in previous work and (iii) the maximum processor performance to (iv) the change point algorithm presented in this paper. For this purpose we use six audio clips totaling 653 seconds of audio, each running at a different set of bit and sample rates as shown in Table 3.1. We have found that there was very little variation on frame-by-frame basis in decoding rate within a given audio clip, but the variation in decoding rate between clips can be large as shown in Table 3.1 (the decoding rates are for 202.4MHz processor frequency).

Table 3.1: MP3 audio streams

MP3 Audio Clip Label	Bit rate (Kb/s)	Sample Rate (KHz)	Decoding Rate (frames/s)
A	16	16	51.35
B	16	32	27.30
C	32	16	49.80
D	32	32	26.05
E	64	16	47.95
F	64	32	25.25

Table 3.2: MP3 audio DVS

MP3 Audio Sequence	Result	Ideal	Change Point	Exp. Ave.	Max
ACEFBD	Energy	196	217	225	316
	Fr.Delay	0.1	0.09	0.1	0
BADECF	Energy	189	199	231	316
	Fr.Delay	0.1	0.09	0.1	0
CEDAFB	Energy	190	214	232	316
	Fr.Delay	0.1	0.04	0.1	0

During decoding, the DVS algorithm detects changes in both arrival and decoding rates for the MP3 audio sequences. The resulting energy (kJ) and average total frame delay (s) are displayed in Table 3.2. Each sequence consists of a combination of six audio clips. For all sequences, the frame arrival rate varied between 16 and 44 frames/sec. Our change point algorithm performs well, its results are very close to the ideal, with no performance loss as compared to the ideal detection algorithm that allows an average $0.1s$ total frame delay (corresponding to 6 extra frames of audio in the buffer).

The next set of results are for decoding two different video clips. In contrast to MP3 audio, for MPEG video there is a large variation in decoding rates on frame-by-frame basis (this has been shown in [5, 13] as well). We again report results for the ideal detection, the exponential average, the maximum processor performance and our change point algorithm. The ideal detection algorithm allows for 0.1s average total frame delay equivalent to 2 extra frames of video in the buffer. The arrival rate varies between 9 and 32 frames/second. Energy (kJ) and average total frame delay (s) are shown in Table 3.3. The results are

Table 3.3: MPEG video DVS

MPEG Video Clip	Result	Ideal	Change Point	Exp. Ave.	Max
Football (875s)	Energy Fr.Delay	214 0.1	218 0.11	300 0.16	426 0
Terminator2 (1200s)	Energy Fr.Delay	280 0.1	294 0.11	385 0.16	570 0

similar to MP3 audio. The exponential average shows poor performance and higher energy consumption due to its instability (see Figure 3.9). Our change point algorithm performs well, with significant savings in energy and a very small performance penalty (0.11s frame delay instead of allowed 0.1s).

Table 3.4: DPM and DVS

Algorithm	Energy (kJ)	Factor
None	4260	1.0
DVS	3142	1.4
DPM	2460	1.7
Both	1342	3.1

Finally, we combine the dynamic voltage scaling detection with power management algorithms presented in [73, 74]. We use a sequence of audio and video clips, separated by idle time. During longer idle times, the power manager has the opportunity to place the SmartBadge in the standby state. The optimal power management policy can be obtained by either of the two approaches presented in [73, 74] as the only decision point is upon the entrance into the idle state. Table 3.4 shows the energy savings if we implemented only dynamic voltage scaling (and thus did not transition into standby state during longer idle times), or if only power management is implemented (and thus processor runs at maximum frequency and voltage in the active state) and finally also for the combination of the two approaches. We obtain savings of a factor of three when expanding the power manager to include dynamic voltage scaling with our change point detection algorithm.

3.5 Summary

A new approach for dynamic voltage scaling that can be used as a part of a power managed system, such as systems presented in Chapter 2, has been presented in this chapter. The dynamic voltage scaling algorithm is consists of two tasks: (i) change point detection algorithm that detects the change in arrival or decoding rates, and (ii) the frequency setting policy that sets the processor frequency and voltage based on the current arrival and decoding rates in order to keep constant performance. I tested the algorithm on MPEG video and MP3 audio running on the SmartBadge portable device. The change point detection algorithm is very stable as compared to the exponential moving average algorithm presented previously. As a result, it gives large energy savings at a small performance penalty for both MPEG video and MP3 audio applications. Finally, I implemented the DVS algorithm together with power management algorithms and showed **factor of three savings in energy** due to the combined approach.

Chapter 4

Energy Efficient Hardware Design

4.1 Introduction

The overall objective of this thesis is to improve energy efficiency of systems with new design and utilization techniques. In the previous two chapters I introduced two techniques aimed at improving energy utilization of systems. Since dynamic power management and dynamic voltage scaling can only improve how the system is used, there is a need for a methodology that enables energy efficient design of hardware and software. In this chapter I will present a methodology for energy efficient design of hardware, and in the next chapter I will address the design of software.

Energy consumption and power dissipation are critical factors in system design. Peak power dissipation sets constraints on thermal and power supply design for the system. Average power consumption is directly related to battery life, hence it may be the critical factor that sets system weight and cost. CAD tool support is needed to evaluate performance and energy consumption in system designs.

The primary motivation for this work comes from the experience with the redesign of a SmartBadge. The SmartBadge is best described with a system model consisting of a microprocessor with level-1 (L1) cache, off-chip memory and DC-DC converter connected with the interconnect, which also represents the basic configuration of many electronic systems. The design task was to enhance the prototype implementation of the SmartBadge by adding other components such as real-time MPEG video decode. The original hardware

did not meet either the performance or the energy consumption constraints when running the MPEG decode algorithm. As a result, both the hardware and the software architectures had to be redesigned, while keeping the energy consumption under tight control.

Design process for such a system starts with the selection of the commodity components that may meet the performance and the energy consumption criteria based on the data sheets. Typically only a few processor families can be evaluated due to resource and time limitations. In addition, many companies often license an architecture and as a result prefer to focus designs on the processor family licensed. In this example, the ARM processor family [1] was selected to illustrate the methodology for cycle-accurate energy consumption simulation used in the design of the SmartBadge.

Cycle-accurate instruction-level simulators are used for performance estimation of the software portion of the design industry-wide. Whole system evaluation is often done on prototype boards. Due to long design times and costs for prototype board design, only a few hardware architectures can be tried. Moreover, power dissipation measurements require board instrumentation: a time-consuming and error-prone process. The advantages of the simulation-based methodology are that it is easy to explore many different hardware and software architectures and thus obtain accurate performance and energy consumption estimates. As a result, I extended the basic instruction-level simulator provided in the ARM software development kit with the cycle-accurate energy consumption models for the processor with the level 1 and the level 2 caches, the off-chip memories, the interconnect, the DC-DC converter and the battery. The methodology presented in this work can be applied to any cycle-accurate instruction-level simulator.

The rest of this chapter is organized as follows. The implementation of cycle-accurate energy consumption simulator is presented in Section 4.2. Section 4.3 shows that the simulation results of timing and energy dissipation using the methodology presented are within 5% of the hardware measurements for the Dhrystone test case. Hardware architecture trade-offs for SmartBadge's real-time MPEG video decode design are explored using cycle-accurate energy simulation in Section 4.4.

4.2 Cycle-Accurate Energy Consumption Simulator Implementation

The system used in this work to illustrate my methodology, the SmartBadge, has an ARM processor. As a result, I implemented the energy models as extensions to the cycle-accurate instruction-level simulator for the ARM processor family, called the ARMmulator [1]. The ARMulator is normally used for functional and performance validation. Figure 4.1 shows the simulator architecture. The typical sequence of steps needed to set up system simulation can be summarized as follows. (i) The designer provides a simple functional model for each system component other than the processor. (ii) The functional model is annotated with a cycle-accurate performance model. (iii) Application software (written in C) is cross-compiled and loaded in specified locations of the system memory model. (iii) The simulator runs the code and the designer can analyze execution using a cross-debugger or collecting statistics. A designer interested in using our methodology would only need to additionally provide cycle-accurate energy models for each component during step (ii) of the simulation setup. Thus, the designer can obtain power estimates with little incremental effort.

We developed a methodology for enhancing cycle-accurate simulator with energy models of typical components used in embedded system design. Each component is characterized with equivalent capacitance for each of its power states. Energy spent per cycle is a function of equivalent capacitance, current voltage and frequency. The equivalent capacitance allows us to easily scale energy consumed for each component as frequency or voltage of operation change. Equivalent capacitances are calculated given the information provided in data sheets.

Internal operation of our simulator proceeds as follows. On each cycle of execution the ARMulator sends out the information about the state of the processor ("cycle type") and its address and data busses. Two main classes of processor cycle types are *processor active*, where active power is consumed, and *processor idle*, where idle power is consumed. The processor idle state represents an off-chip memory request. The number of cycles that the processor remains idle depends on L2 cache and memory model access times. L2 cache, when present, is always accessed before the main memory and so is active on every memory access request. On L2 cache miss, main memory is accessed. Memory

model accounts for energy spent during the memory access. The interconnect energy model calculates energy consumed by the interconnect and pins based on the number of lines switched during the cycle on the data and address busses. The DC-DC converter energy model sums all the currents consumed each cycle by other system components, accounts for its efficiency loss, and gets the total energy consumed from the battery. The battery model accounts for battery efficiency losses due to the difference between the rated current and discharge current computed the current cycle.

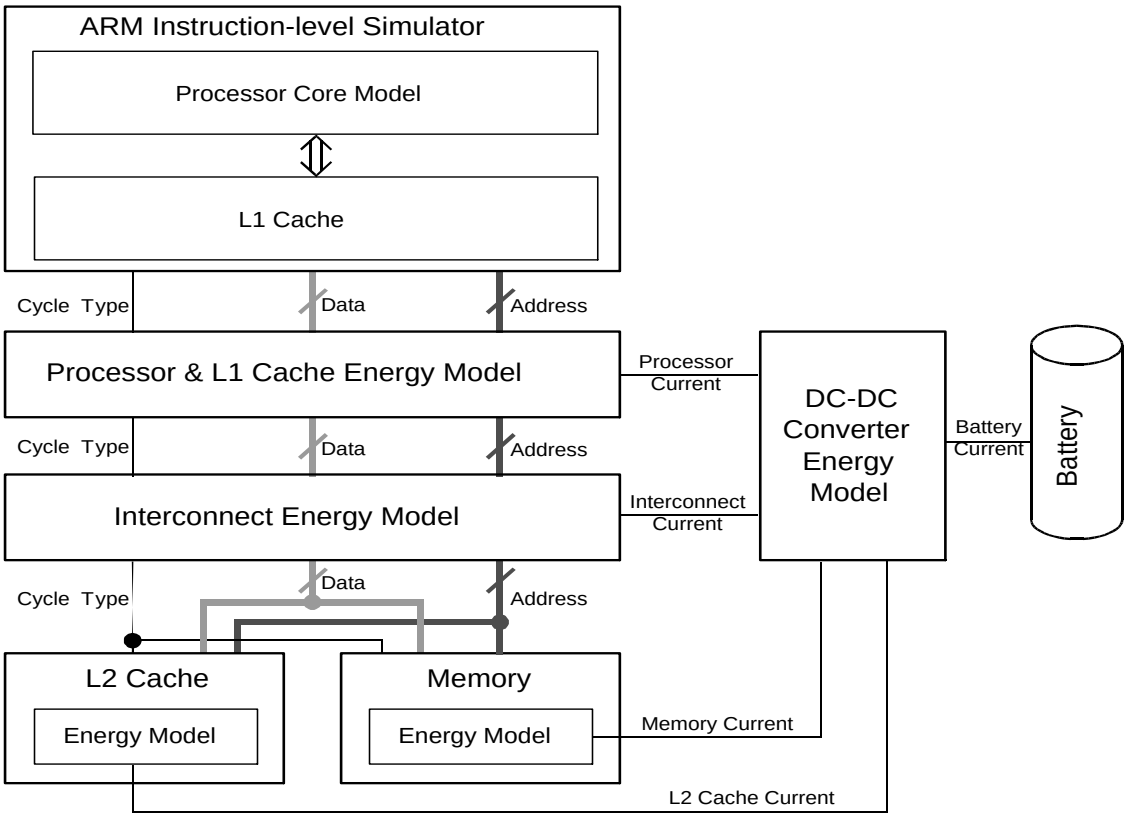

Figure 4.1: Simulator Architecture

The total energy consumed by the system per cycle is the sum of energies consumed by the processor and L1 cache (E_{CPU}), interconnect and pins (E_{Line}), memory (E_{Mem}), L2 cache (E_{L2}), the DC-DC converter (E_{DC}) and the efficiency losses in the battery ($E_{Bat.}$):

$$E_{Cycle} = E_{CPU} + E_{Line} + E_{Mem} + E_{L2} + E_{DC} + E_{Bat} \tag{4.1}$$

The total energy consumed during the execution of the software on a given hardware architecture is the sum of the energies consumed during the each cycle. Models for energy consumption and performance estimation of each system component are described in the following sections.

4.2.1 Processor

The ARM simulator provides a cycle-accurate, instruction-level model for ARM processors and L1 on-chip cache. The model was enhanced with energy consumption estimates based on the information provided by the data sheets. Two power states are considered: active state in which processor is running with the on-chip cache, and the state in which the processor is executing NOPs while waiting to fill the cache.

Note that in the case of StrongARM processor used in this work, the data sheet values for current consumption correspond well to the measured values. Wan [84] extended StrongARM processor model with base current costs for each instruction. The average power consumption for most of the instructions is 200mW measured at 170MHz. Load and store instructions required 260mW each. Because the difference in energy per instruction is minimal, it can be expected that the average power consumption value from the data sheets is on the same level of accuracy as the instruction-level model. Thus we can use data sheet values to derive equivalent capacitances for the StrongARM. Note that for other processors data sheet values would need to be verified by measurement, as often data sheet values report the maximum power consumption, instead of typical.

When the processor is executing with the on-chip cache, it consumes the active power specified in the data sheet P_m measured at given voltage V_m and frequency of operation f_m. Total equivalent active capacitance within the processor, $C_{CPU,a}$, is estimated as:

$$C_{CPU,a} = \frac{P_m}{V_m^2 f_m} \tag{4.2}$$

The amount of energy consumed by processor and L1-cache at specified processor cycle

time T_{cycle} and CPU core voltage V_{cc} is:

$$E_{CPU,active} = P_{CPU,a} T_{cycle} = C_{CPU,a} V_{cc}^2 \tag{4.3}$$

When there is an on-chip cache miss, the processor stalls and executes NOP instructions which consume less power. $C_{CPU,NOP}$ can be estimated from the power consumed during execution of NOPs $P_{CPU,NOP}$ at voltage V_m and frequency f_m:

$$C_{CPU,NOP} = \frac{P_{CPU,NOP}}{V_m^2 f_m} \tag{4.4}$$

The energy consumed within processor core per cycle while executing NOPs is:

$$E_{CPU,NOP} = C_{CPU,NOP} V_{cc}^2 \tag{4.5}$$

4.2.2 Memory and L2 cache

The processor issues an off-chip memory access when there is a L1 cache miss. The cache-fill request will either be serviced by the L2 cache if one is present in the design or directly from the main memory. On L2 cache miss, a request is issued to the processor to fetch data from the main memory. Data sheets specify the memory and L2 cache access times, and energy consumed during active and idle states of operation.

Memory access time, T_{mem}, is scaled by the processor cycle time, T_{cycle}, to obtain the number of cycles the processor has to wait to serve a request, N_{wait} (Equation 4.6). Wait cycles are defined for two different types of memory accesses: sequential and non-sequential. Sequential access is at the address immediately following the address of the previous access. In burst type memory the sequential access is normally a fraction of the first, non-sequential, access.

$$N_{wait} = \frac{T_{mem}}{T_{cycle}} \tag{4.6}$$

Two energy consumption states are defined for each type of memory: active and idle. Energy consumed per cycle while memory is in active state operating at supply voltage V_{dd} is

a function of equivalent active capacitance, voltage of operation and number of total access cycles ($N_{wait}+1$):

$$E_{Mem,active} = \frac{C_{mem}V_{dd}^2}{N_{wait}+1} \tag{4.7}$$

Active memory capacitance, C_{mem}, can be estimated from the active power specified in the data sheet, P_{mem}, measured at voltage V_m and frequency f_m:

$$C_{mem} = \frac{P_{mem}}{V_m^2 f_m} \tag{4.8}$$

Multibank memory can be represented as multiple one-bank memories.

Idle state can be further subdivided into multiple states that describe modes of operation for different types of memories. For example, DRAM might have two idle states: refresh and sleep. The designer specifies the percentage of the time ρ_i memory spends in each idle state. Total idle energy per cycle for memory is:

$$E_{Mem,idle} = T_{cycle}\sum_{i=0}^{n} P_i\rho_i \tag{4.9}$$

where P_i is power consumption in idle state i. Both RAM and ROM are represented with the same memory model, but with different parameters.

The L2 cache access time and energy consumption are treated the same way as any other memory. L2 cache organization is determined from the number of banks, lines per bank, and words per line. Line replacement can follow any of the well-known replacement policies. Cache hit rate is strongly dependent on its organization, which in turn affects the total memory access time and the energy consumption. Note that we are simulating details of the L2 cache access, and thus know the exact L2 cache miss rate.

4.2.3 Interconnect and Pins

The interconnects on the PCB can contribute a large portion of the off-chip capacitance. Capacitance per unit length of the interconnect is a parameter in the energy model that can be obtained from the PCB manufacturer. The length of an interconnect can be estimated by

the designer based on the approximate placement of the selected components on the PCB. Pin capacitance values are reported on the data sheets.

For each component the average length of the clock line, data and address buses between the processor and the component are provided as one of the input simulation parameters. Hence, the designer is free to use any wire-length estimate [20] or measurement. The interconnect lengths used in our simulation of SmartBadge come from the prototype board layout.

The total capacitance switched during one cycle is shown in Equation 4.10. It depends on the capacitance of one interconnect line and the pins attached to it, C_{switch}, and the number of lines switched during the cycle, N_{switch}.

$$C_{line} = N_{switch} C_{switch} \tag{4.10}$$

The total energy consumed per cycle, $E_{Interconnect}$, is a function of the voltage swing on the lines that switched, V_{dd}, total capacitance switched, C_{line}, and the total time to access the memory, $N_{wait} + 1$:

$$E_{Line} = \frac{C_{line} V_{dd}^2}{N_{wait} + 1} \tag{4.11}$$

4.2.4 DC-DC Converter

DC-DC converter losses can account for a significant fraction of the total energy consumption. Figure 4.2 from the datasheets shows the dependence of efficiency on the DC-DC converter output current. Total current drawn from the DC-DC converter by the system each cycle, I_{out}, is a sum of the currents drawn by each system component. A component current, I_c, is defined by:

$$I_c = \frac{E_c}{V_c T_{cycle}} \tag{4.12}$$

where E_c is the energy consumed by the component during cycle of length T_{cycle} at operating voltage V_c.

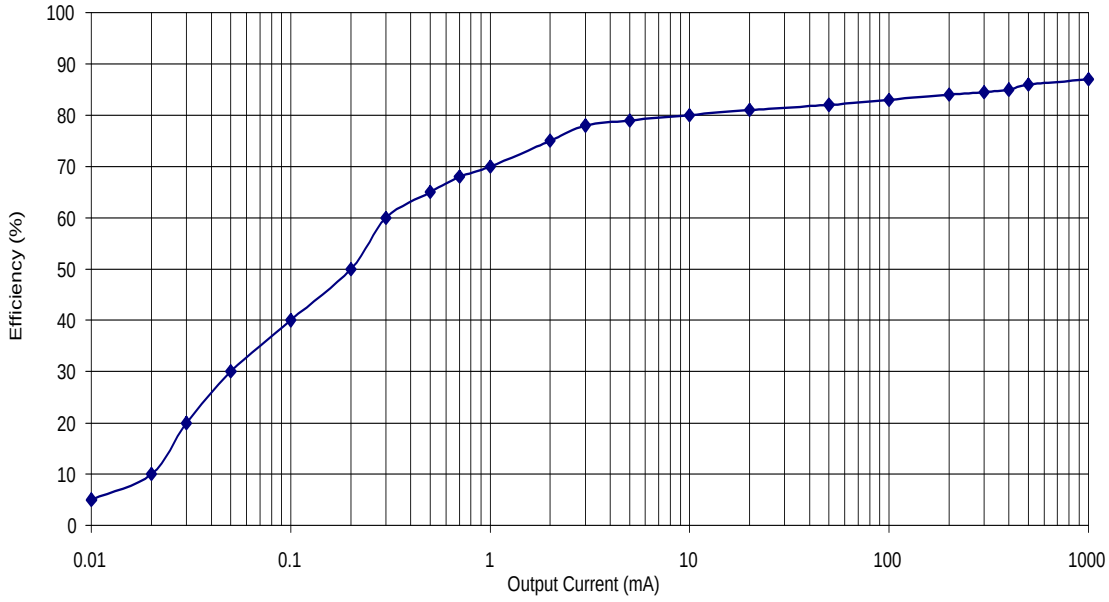

Figure 4.2: DC-DC Converter Efficiency

Total current drawn from the battery, I_{bat} can be calculated as:

$$I_{bat} = \frac{I_{out}}{\eta_{DC}} \tag{4.13}$$

Efficiency, η_{DC}, can be estimated using linear interpolation from the data points derived from the output current versus efficiency plot in the data sheet. From our experience, a table with 20 points derived from the data sheets gives enough information for accurate linear estimation of values not directly represented in the table.

Total energy DC-DC converter draws from the battery each cycle is:

$$E_{DCbat} = I_{bat} V_{bat} T_{cycle} \tag{4.14}$$

The energy consumed by the DC-DC converter, E_{DC}, is difference between the energy provided by the battery, E_{DCbat} and the energy consumed from the DC-DC converter by all

other components, E_{out}:

$$E_{DC} = E_{DCbat} - E_{out} \tag{4.15}$$

4.2.5 Battery Model

The main battery characteristic is its rated capacity measured in mWhr. Since total available battery capacity varies with the discharge rate, manufacturers specify plots in the datasheets with discharge rate versus battery efficiency similar to the one shown below.

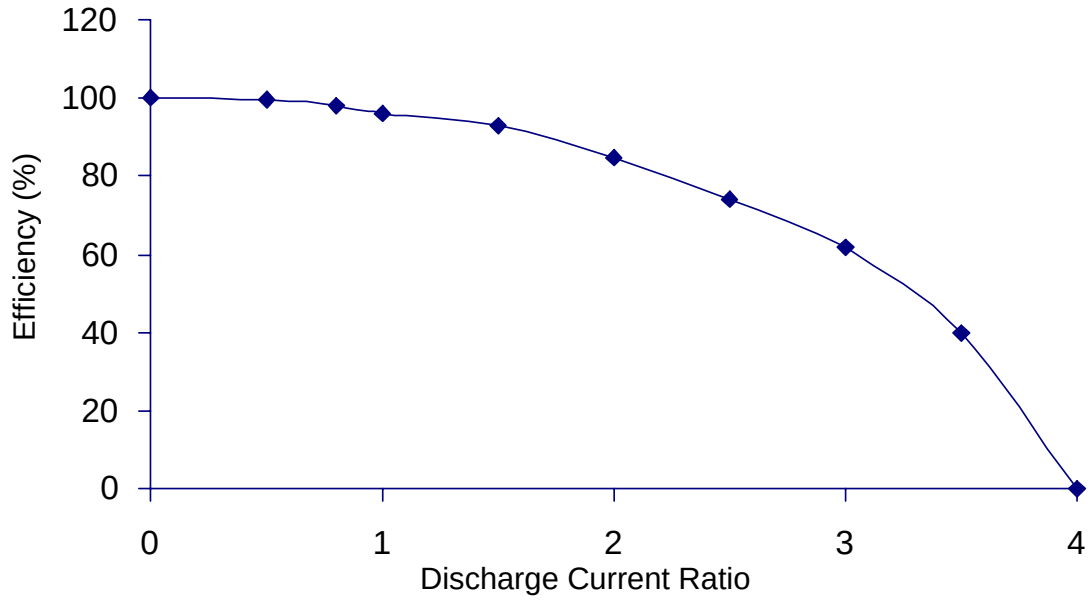

Figure 4.3: Battery Efficiency

The discharge rate (or discharge current ratio) is given by:

$$R_I = \frac{I_{ave}}{I_{rated}} \tag{4.16}$$

where I_{rated}, the rated discharge current, is derived from the battery specification and I_{ave} is the average current drawn by the DC-DC converter. As battery cannot respond to instantaneous changes in current, a first order time constant τ is defined to determine the short-term

average current drawn from the battery [88]. Given τ, and processor cycle time T_{cycle}, we can compute N_{bat}, the number of cycles over which average DC-DC current is calculated:

$$N_{bat} = \frac{\tau}{T_{cycle}} \tag{4.17}$$

then, I_{ave} is computed as:

$$I_{ave} = \frac{1}{N_{bat}} \sum_{cycle=1}^{N_{bat}} I_{system}(cycle) \tag{4.18}$$

where I_{system} is the instantaneous current drawn from the battery. With discharge current ratio, we estimate battery efficiency using battery efficiency plot such as the one shown in Figure 4.3. The total energy loss of the battery per cycle, E_{Bat}, is the product of energy drained from the battery by the system with the efficiency loss $(1 - \eta_{Bat})$:

$$E_{Bat} = (1 - \eta_{Bat}) I_{ave} V_{Bat} T_{cycle} \tag{4.19}$$

Given the battery capacity model described above, battery estimation is performed as follows. First, the designer characterizes the battery with its rated capacity, the time constant and the table of points describing the discharge plot similar to the one shown in Figure 4.3. During each simulation cycle discharge current ratio is computed from the rated battery current and average DC-DC current calculated from the last N_{bat} cycles. Efficiency is calculated using linear interpolation between the points from the discharge plot. Total energy drawn from the battery during the cycle is obtained from Equation 4.19. Lower efficiency means that less battery energy remains and thus the battery lifetime is proportionally lower. For example, if battery efficiency is 60% and its rated capacity is $100mAhr$ at $1V$, then the battery would be drained in 12 minutes at average DC-DC current of $300mA$. With efficiency of 100% the battery would last 1 hour.

4.3 Validation of the Simulation Methodology

We validated the cycle-accurate power simulator by comparing the computed energy consumption with measurements on the SmartBadge prototype implementation. The SmartBadge prototype consists of the StrongARM-1100 processor, DC-DC Converter, FLASH and SRAM on a PCB board. All the components except the CPU core are powered through the 3.3V supply line. CPU core runs on 1.5V supply. DC-DC converter is powered by the 3.5V supply. DC-DC converter efficiency table contains 22 points derived from the plot shown in Figure 4.2. Stripline interconnect model is used with $1.6pF/cm$ capacitance calculated based on the PCB board characteristics [58]. Table 4.1 shows other system components. Average current consumed by the processor's power supply and the total current drawn from the battery are measured with digital multimeters. Execution time is measured using the processor timer.

Table 4.1: Dhrystone Test Case System Design

Component Units	Cycle T. (ns)	Active P (mW)	Idle P (mW)	Pin Cap. (pF)	Line L. (cm)
SA-1100	5-20	400	170	5	N/A
FLASH (1MB)	80	74	0.5	10	2
SRAM (1MB)	90	55	0.01	8	3

Industry standard Dhrystone benchmark is used as a vehicle for methodology verification. Measurements and simulations have been done for ten different operating frequencies of SA-1100 and SA-110 processors. Dhrystone test case is run 10 million times, 445 instructions per loop. Simulations ran on HP Vectra PC with Pentium II MMX 300 MHz processor and 128 MB of memory. Hardware ran 450 times faster than the simulations without the energy models. Simulations with energy models were slightly slower (about 7%). Figure 4.4 show average processor core and battery currents. Average simulation current is obtained by dividing the total energy consumed by the processor core or the battery with their respective supply voltages and the total execution time.

Simulation results are within 5% of the hardware measurements for the same frequency

of operation. The methodology presented in this paper for cycle-accurate energy consumption simulation is very accurate and thus can be used for architecture design exploration in embedded system designs. An example of such exploration is presented next.

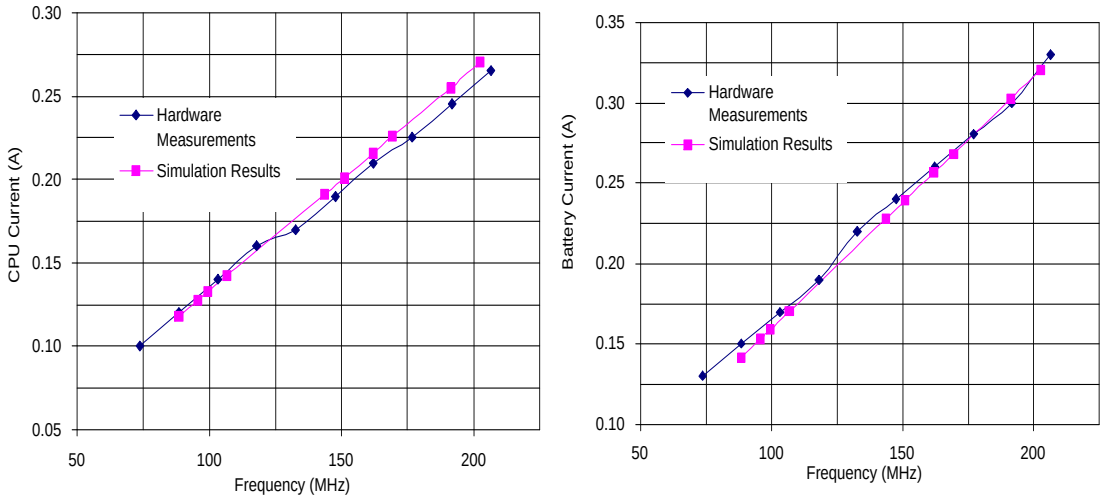

Figure 4.4: Average Processor Core and Battery Currents

4.4 Embedded MPEG Decoder System Design Exploration

The primary motivation for the development of cycle-accurate energy consumption simulation methodology is to provide an easy way for embedded system designers to evaluate multiple hardware and software architectures with respect to performance and energy consumption constraints. In this section we will present an application of the simulation methodology to embedded MPEG video decoder system design exploration. The MPEG decoder design consists of the processor, the off-chip memory, the DC-DC converter, output to the LCD display, and the interface to the source of the MPEG stream. The input and output portions of the MPEG decoder design will not be considered at this point. We focus on selection of memory hierarchy that is most energy efficient.

The characteristics of memory components considered are shown in Table 4.2. Two

Table 4.2: Memory Architectures for MPEG Design

Name	First Acc. (ns)	Burst Acc. (ns)	Actjve Pwr (mW)	Idle Pwr (mW)	Line Cap. (pF)	Pin Cap. (pF)	Manuf.
FLASH	80	N/A	75	0.5	4.8	10	Intel
BFLASH	80	40	600	2.5	4.8	10	TI
SRAM	90	N/A	185	0.1	8	8	Toshiba
BSRAM	90	45	365	1.7	8	8	Micron
BSDRAM	30	15	430	10	8	8	Micron
L2 cache	20	10	1985	330	3.2	5	Motorola

different instruction memories were evaluated – low-power FLASH and power-hungry burst FLASH. We looked at three different data memories – low-power SRAM, faster burst SRAM and very power-hungry burst SDRAM. Both instruction and data memories are 1MB in size. We considered using L2 cache in addition to L1 cache. Unified L2 cache is 256Kb, 4-way set associative. The hardware configurations simulated are shown in Table 4.3. The MPEG video decode sequence we used has 12 frames running at 30 frames/second, with two I, three P and seven B-frames. We found that the results we obtained with a shorter video sequence matched well the results obtained with the longer trace.

Table 4.3: Hardware Configurations

Name	Instruction Memory	Data Memory	L2 cache Present
Original	FLASH	SRAM	no
L2 cache	FLASH	BSDRAM	yes
Burst SRAM	BFLASH	BSRAM	no
Burst SDRAM	BFLASH	BSDRAM	no

Figure 4.5 shows the amount of time each system component is active during the MPEG decode and the amount of energy consumed. The original configuration is limited by the bandwidth of data memory. L2 cache is very fast, but also consumes too much energy. Burst SDRAM design fully solves the memory bandwidth problem with least energy consumption. Instruction memory constitutes a very small portion of the total energy due to

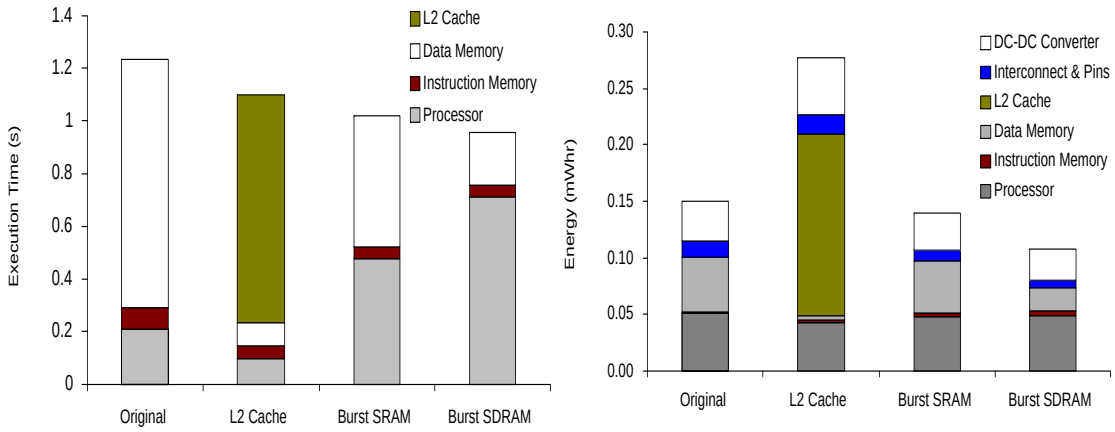

Figure 4.5: Performance and energy consumption for hardware architectures

the relatively large L1 cache in comparison to the MPEG code size. The DC-DC converter consumes a significant amount of total energy and thus should be considered in system simulations. We conclude from this example that using faster and more power-hungry memory can be energy efficient.

The analysis of peak energy consumption and the fine tuning of the architectures can be done by studying the energy consumption and the memory access patterns over a period of time. Figure 4.6 shows the energy consumption over time of the processor with burst FLASH and SRAM. Peak energy consumption can reach twice the average consumption, so the thermal characteristics of the hardware design, the DC-DC converter and the battery have to be specified accordingly.

For best battery utilization, it is important to match the current consumption of the embedded system to the discharge characteristic of the battery. On the other hand, the more capacity battery has, the heavier and more expensive it will be. Figure 4.7 shows that the instantaneous battery efficiency varies greatly over time with MPEG decode running on the hardware described above.

Lower capacity batteries have larger efficiency losses. Figure 4.8 shows that the total decrease in battery lifetime when continually running MPEG algorithm on a battery with lower rated discharge current can be as high as 16%. The battery's time constant was set to $\tau = 1ms$.

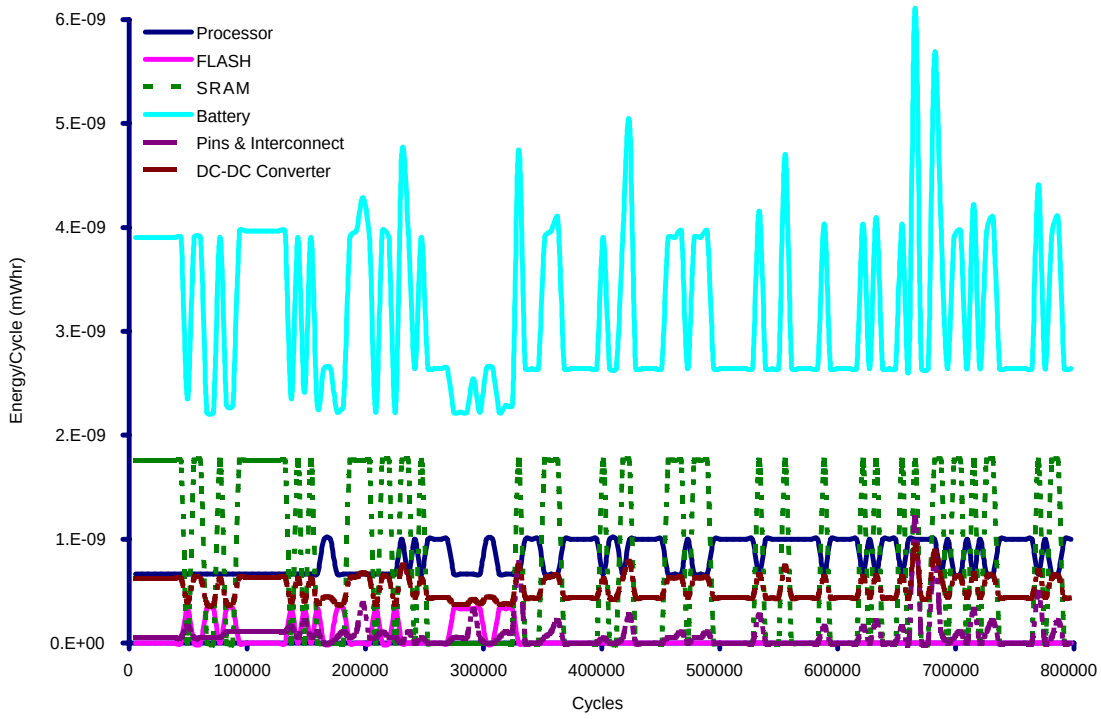

Figure 4.6: Cycle-accurate Energy Plot

The design exploration example presented in this section illustrates how the methodology for cycle-accurate energy consumption simulation can be used to select and fine-tune hardware configuration that gives the best trade-off between performance and energy consumption.

The main limitation of cycle-accurate energy simulator is that the impact of code optimizations is not easily evaluated. For example, in order to evaluate energy efficiency of two different implementations of a particular portion of software, the designer would need to obtain cycle-by-cycle plots and then manually relate cycles to the software portion of interest. The profiling methodology presented next addresses this limitation.

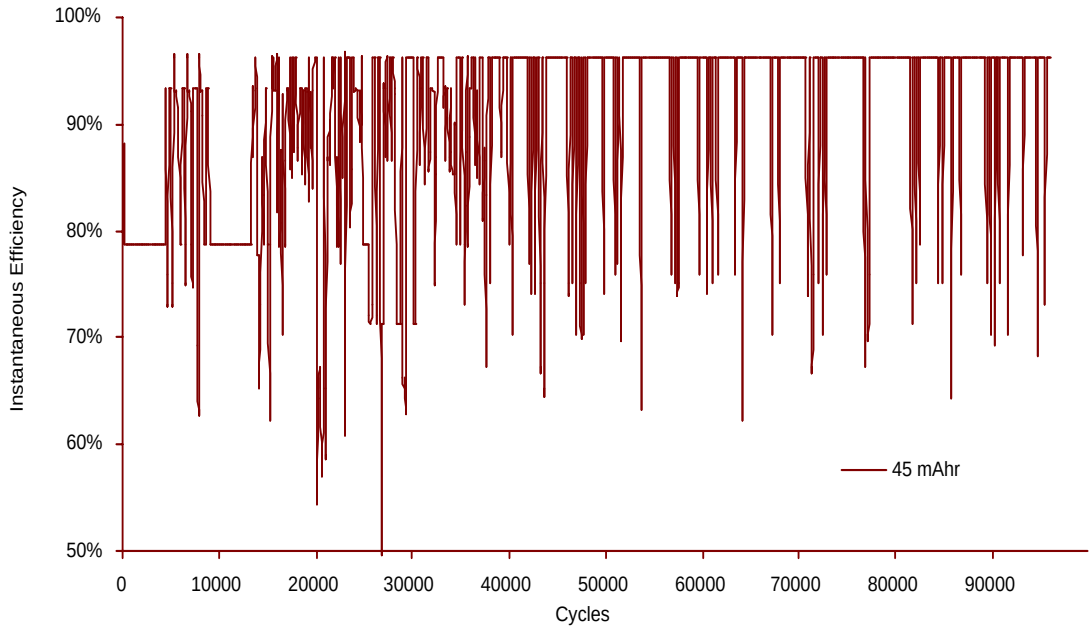

Figure 4.7: Battery Efficiency for MPEG Decoder

4.5 Summary

A methodology for cycle-accurate simulation of performance and energy consumption in embedded systems has been presented in this chapter. Accuracy, modularity and ease of integration with the instruction-level simulators widely used in industry make this methodology very applicable to the embedded system hardware and software design exploration.

Dhrystone benchmark has been used to verify accuracy of the energy and the performance estimates. Simulation is found to be within 5% of the hardware measurements. MPEG video decoder design exploration has been presented as an example of how the methodology can be used in practice to aid in the selection of the best hardware and software configuration.

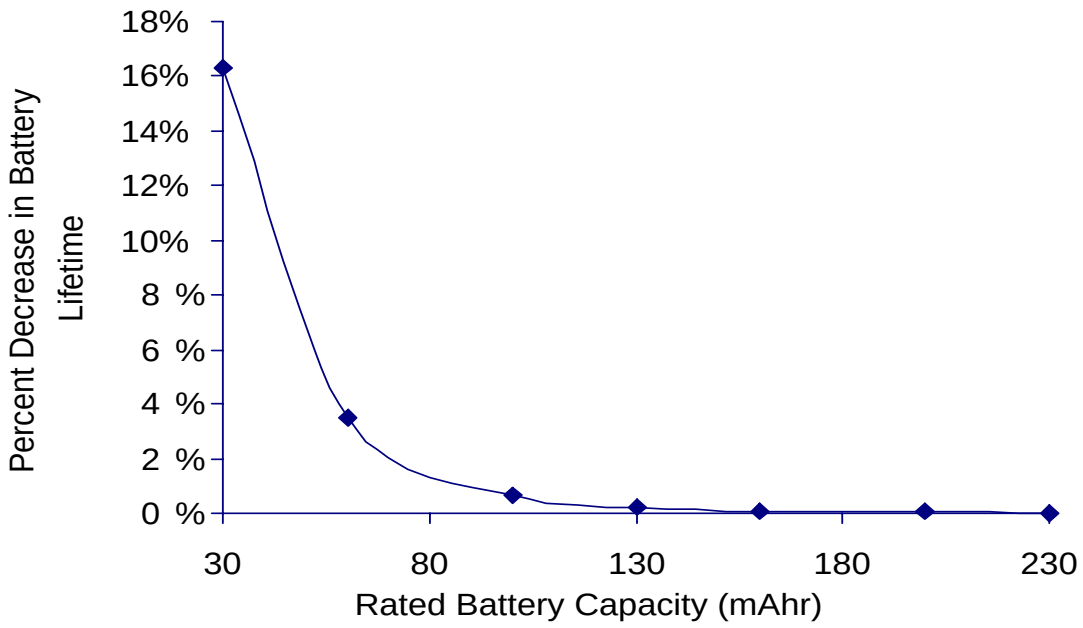

Figure 4.8: Percent Decrease in Battery Lifetime for MPEG Decoder

Chapter 5

Energy Efficient Software Design

5.1 Introduction

In an industrial environment, the degrees of freedom in hardware design are often very limited but for software a lot more freedom is available. As a result, a primary requirement for system-level design methodology is to effectively support code energy consumption optimization. Several techniques for code optimization have been presented in the past [78, 79, 53, 46, 80]. All these techniques focus on automated instruction-level optimizations driven by the compiler.

Unfortunately, currently available commercial compilers have limited capabilities. The improvements gained when using standard compiler optimizations are marginal compared to writing energy efficient source code [70]. The largest energy savings were observed at the inter-procedural level that compilers have not been able to exploit. Thus, it is critical to develop a software design methodology that enables fast and easy manual code redesign.

Code optimization requires extensive program execution analysis to identify energy-critical bottlenecks and to provide feedback on the impact of transformations. Profiling is typically used to relate performance to the source code for CPU and L1 cache [1]. Leveraging the cycle-accurate energy consumption simulator presented in the previous chapter, I implemented a code profiling tool that gives percentages of time and energy spent in each procedure for every system component, not only CPU and L1 cache. Thanks to energy profiling, the programmer can easily identify the most energy-critical procedures, apply

transformations and estimate their impact not only on processor energy consumption, but also on memory hierarchy and system busses.

A profiler is only a tool that can be used during software design and optimization. The profiler is a critical part of a general code transformation methodology I present next. The methodology consists of three categories of source code optimizations: algorithmic changes, data representation changes and instruction-level optimizations. In addition to the general code optimization methodology, I present a series of suggestions in source code writing style that can save from 1% to over 90% of energy developed specifically for the ARM processors. Most of these optimizations can be implemented with small revisions to other processors.

The profiling support is presented in Section 5.2. The general code optimization methodology is described in Section 5.3, and is applied to a full software design example of MP3 audio decoder for the SmartBadge. Extensive experimental results are given. Processor specific code transformations are discussed in detail in Section 5.5. Simulation results for this set of code transformations show large energy savings.

5.2 Profiling software energy consumption

The profiler architecture is shown in Figure 5.1. Shaded portion represents the extension we made to the cycle-accurate energy simulator to enable code profiling. Profiling for energy and performance enables designers to identify those portions of their source code that need to be further optimized in order to either decrease energy consumption, increase performance or both. Our profiler enables designers to explore multiple different hardware and software architectures, as well as to do statistical analysis based on the input samples. In this way the design can be optimized for both energy consumption and performance based on the expected input data set.

The profiler operates as follows. Source code is compiled using a compiler for a target processor (e.g. application or operating system code). The output of the compiler is the executable that the cycle-accurate simulator executes (represented in this figure as assembly code that is input into the simulator) and a map of locations of each procedure in the executable that a profiler uses to gather statistics (the map is correspondence of assembly code

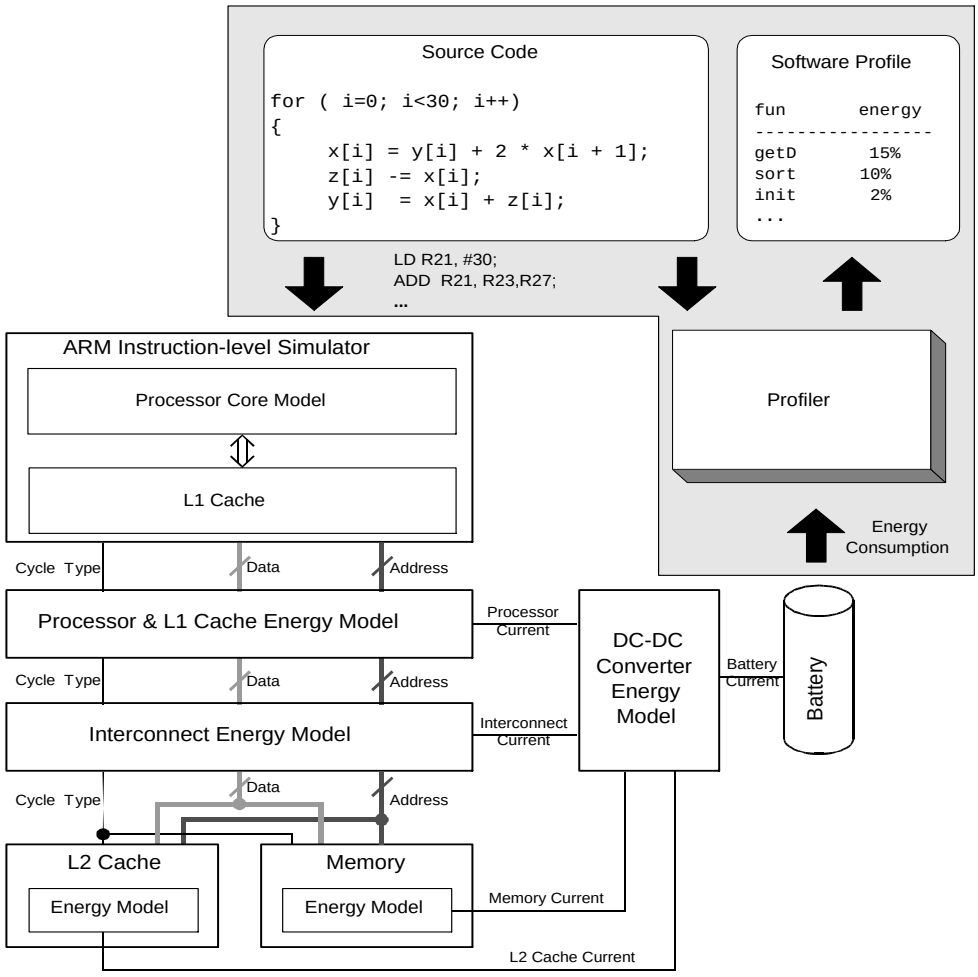

Figure 5.1: Profiler Architecture

blocks to procedures in 'C' source code). In order to increase the simulation speed, a user-defined profiling interval is set, so that the profiler gathers statistics only at predetermined time increments. Usually an interval of $1\mu s$ is sufficient. Note that longer intervals will give slightly faster execution time, with a loss of accuracy. Very short intervals (on the other of a few cycles) have larger calculation overhead. For example, energy consumption calculation gives approximately 10% overhead to standard cycle-accurate performance simulation. Profiling with an interval of $1\mu s$ gives negligible overhead over energy simulation (less then 1%), with still accurate results.

During each cycle of operation, the cycle-accurate energy consumption simulator calculates the current total execution time and energy consumption of all system components as shown in Equation 4.1. The profiler works concurrently with the cycle-accurate simulator. It periodically samples the simulation results (using sample interval specified by the user) and maps the energy and performance to the function executed using information gathered at the compile time. Once the simulation is complete, the results of profiling can be printed out by the total energy or time spent in each function.

Table 5.1: Sample Energy Profiling

Name	Cumulative (mWhr)	Self (mWhr)
main	3.20E-01	2.52E-02
...		
III_hybrid		6.71E-02
SubBandSynthesis		3.72E-02
III_stereo		2.75E-02
III_reorder		2.02E-02
III_antialias		1.45E-02
III_dequantize_sample		1.40E-02
III_hufman_decode		3.74E-03
III_get_scale_factor		1.28E-04
decode_info		3.20E-05
...		
III_hybrid	6.71E-02	6.36E-03
inv_mdctL		6.07E-02
SubBandSynthesis	3.72E-02	1.95E-02
chendct32_scaled		1.77E-02
III_stereo	2.75E-02	2.75E-02
III_reorder	2.02E-02	2.02E-02
III_antialias	1.45E-02	1.45E-02
III_dequantize_sample	1.40E-02	1.40E-02
III_hufman_decode	3.74E-03	1.53E-03
huffman_decoder		2.17E-03
initialize_huffman		1.03E-05
hsstell		3.20E-05

The main advantage of the profiler is that it allows designers to obtain energy consumption breakdown by procedures in their source code after running only one simulation. This

information is of critical importance when designing an embedded system, as it enables designers to quickly identify and address the areas in the source code that will provide largest overall energy savings. A good example of profiler usage is shown in Table 5.1. The table shows a portion of energy profile for MP3 audio decode. The first column gives the name of the top procedure, followed by its children. The next column gives the total energy spent for that procedure. For example, the total energy spent running the program (`main`) is $0.32mWhr$. The final column gives the amount of energy spent only in that particular procedure. For example, under `main` it is clear that `III_hybrid` and its descendants spend the most energy, $0.0671mWhr$. Looking at the entry for `III_hybrid`, it is easy to see that the largest portion of energy is consumed by its child, `inv_mdctL`. Therefore, the procedures to focus optimization on are `inv_mdctL` and `SubBandSynthesis`. Although in this example we showed source code profile of total battery energy consumption, the profiler can report energy consumption for any system component, such as SRAM or the interconnect.

The profiler allows for fast and accurate evaluation of software and hardware architectures. Most importantly, it gives good guidance to the designer during the design process without requiring manual intervention. In addition, the profiler accounts for all embedded system components, not just the processor and the L1 cache as most general-purpose profilers do. In the next section I present a general source code optimization methodology that uses the profiler to guide the code changes. The methodology is used to redesign the MP3 audio decoder running on the SmartBadge.

5.3 General Code Optimization Methodology

Code optimization is the process of translating a high-level specification in an imperative language into optimized machine code for the target processor. Compilers are the tools of choice for code optimization. Extensive research on *optimizing compilers* has been carried out in last few years [55]). Prototype research compilers have shown impressive results [28]. Most optimizing compilers target high-performance and/or general-purpose computers, and relatively little effort has been dedicated to create powerful optimizing

compilers for embedded processors. Even though several researchers are studying automatic code optimization techniques for embedded processors [89], currently, most embedded processors (or DSPs) are programmed directly in assembly by expert programmers and code optimization is mostly based on human intuition and skill.

Our approach to code optimization for embedded systems is to complement the compilers with manual code re-writing and optimization, as the compiler support is still limited. The profiler discussed in the previous section is a critical part of our approach, as it enables fast analysis of source code energy consumption. The main advantage of our approach is that it enables designers to focus first on a very abstract view of the problem, find a good solution, then move down in abstraction, and perform optimizations that are narrower in scope. The complex problem of optimizing an executable specification is partitioned, and its parts are more manageable than the complete problem. In the next subsections, we will describe in detail the three optimization layers defined in out methodology, moving from high to low abstraction. We will illustrate our methodology on optimization of MP3 code [57] for the SmartBadge [51].

5.3.1 Algorithmic optimization

The top layer in the optimization hierarchy targets algorithms. The original specification is first profiled to identify all computational kernels, i.e., the procedures where most time and power are spent. Alternative algorithms for implementing the same functionality are considered and compared with the original one using high-level estimators of algorithmic efficiency (such as number of basic operations). Profiler is utilized in both of these steps. Most promising alternative algorithms are then analyzed in more detail and finally coded. This step is mostly based on human intuition and knowledge, and is unlikely to be automated.

Algorithmic optimizations have high potential, but they also have risks. First, developing and testing algorithms is a time-consuming and error-prone task. Since human resources are always scarce, it is unwise to dedicate too much effort to an activity where success is often based on intuition. Second, asymptotic analysis and operation counts are often misleading as estimators of algorithmic efficiency, hence marginal improvements should be

regarded with suspicion when considering algorithmic changes.

Our approach to algorithmic optimization in MP3 decoding has been conservative. First, we focused on just one computational kernel where a large fraction of run time (and power) was spent, namely the *subband synthesis*. Second, we did not try to develop new original algorithms but we used previously published algorithmic enhancements [29, 30] that are still fully compliant to the MPEG standard. The new algorithm incorporates an integer implementation of the scaled Chen discrete cosine transform (DCT) instead of a generic DCT in the polyphase synthesis filterbank. The use of a scaled DCT reduces the DCT multiply count by 28%.

5.3.2 Data optimization

At a lower level of abstraction than the algorithmic level, we can optimize code by changing the representation of the data manipulated by the algorithms. The main objective is to match the characteristics of the target architecture with the processed data. Signal processing algorithms are often specified by assuming double-precision floating point data to avoid overflows and keep accuracy under control.

Floating point computations are usually more complex and power-hungry than their integer counterparts [81]. As no hardware floating point support is available in the ARM SA-1100 and the MPEG decoder specification performed most computations using doubles, we tried to emulated floating point using ARM's software library. The direct implementation of the decoding algorithm, even after algorithmic optimization, was unacceptably slow and power-consuming.

To overcome this problem, we developed a fixed-precision library and we implemented all computational kernels of the algorithm using fixed precision numbers. The number of decimal digits can be set at compile time. The ARM architecture is designed to support computation with 32-bits integers with maximum efficiency. Little can be gained by reducing data size below 32 bits. On the other hand, when multiplying two 32-bit numbers, the result is a 64-bit number and directly truncating the result of a multiplication to 32 digits frequently leads to incorrect results because of overflow. To increase robustness, 64-bit numbers have been used for fixed-point computation. This data type is supported by

the ARM compiler through the definition of a `long long` integer type. Computing with `long long` integers is less efficient than using 32-bit integers, but results are accurate and the risk of overflow is minimized.

Data optimization produced significant energy savings and speedups for computational kernels of MP3 without any perceivable degradation in quality. The fixed-point library developed for this purpose contains macros for conversion from fixed-point to floating point, accuracy adjustment and elementary function computation. This optimization did not require extensive code rewriting, and it was implemented independently from algorithmic optimization.

5.3.3 Instruction flow optimization

The third layer of optimizations targets low-level instruction flow. After extensive profiling, the most critical loops are identified and carefully analyzed. Source code is then re-written to make computation more efficient. Well-known techniques such as loop merging, unrolling, software pipelining, loop invariant extraction, etc. [55, 4] have been applied. In the innermost loops, code can be written directly as inline assembly, to better exploit specialized instructions.

Instruction flow optimizations have been extensively applied in the MP3 decoder, obtaining significant speedup. We do not describe these optimizations in detail because they are common knowledge in the optimizing compilers literature [55, 4]. However, in our case most optimizations were performed manually due to lack of support by the ARM compiler.

A simple example of this class of transformation is the use of the multiply-accumulate instruction (`MLAL`) available in the ARM SA-1100 core. The inner loops of subband synthesis and inverse modified cosine transform (the two key computational kernels of MP3 decoder), contain matrix multiplications which can be implemented efficiently with multiply-accumulate. In this case, we forced the ARM compiler to use the `MLAL` instruction by inlining it in assembly.

5.3.4 General Code Methodology Summary

We described three code optimization layers that have been useful to optimize MP3 decoding. We found that layering optimizations for decreasing levels of abstraction, and working on each level separately, was a very effective way to tackle the non-trivial task of speeding up and reducing the energy consumed in executing the original specification by more than an order of magnitude. In principle, stepwise optimization may reduce optimality. In practice, it often helps in finding better heuristic solutions in a shorter time. Many of the optimizations we applied manually could be automated, even though automation becomes more problematic as the level of abstraction raises. During code optimization, tool support was essential: code profiling was by far the most useful source of information to direct optimization, and assess its impact.

5.4 Optimizing MP3 audio decoder

The block diagram of the MPEG Layer III audio decoding algorithm (MP3) is shown in Figure 5.2. It consists of three blocks: frame unpacking, reconstruction, and inverse mapping. The first step in decoding is synchronizing the incoming bitstream and the decoder. Huffman decoding of the subband coefficients is performed before requantization. Stereo processing, if applicable, occurs before the inverse mapping which consists an inverse modified cosine transform (IMDCT) followed by a polyphase synthesis filterbank. We obtained the original MP3 audio decoder software from the International Organization for Standardization [37]. Our design goal was to obtain real-time performance with low energy consumption while keeping in full compliance with the MPEG standard.

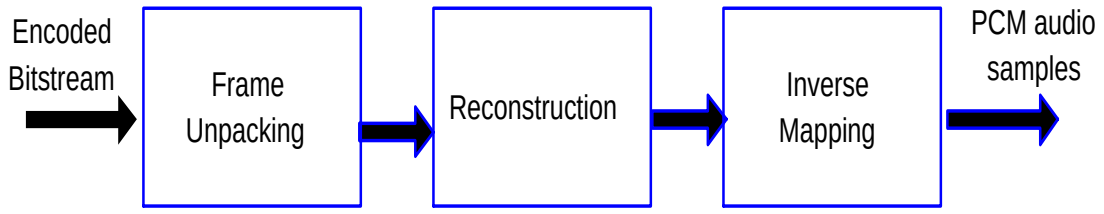

Figure 5.2: MP3 Audio Decoder Architecture

Table 5.2 shows the top three functions in energy consumption for each code revision we worked on. The original code has a very large overhead due to floating point emulation - about 80% of energy consumption. The next largest issue is the redesign of SubBandSynthesis function that implements the polyphase synthesis filterbank. The details of each optimization type, namely algorithmic, data and instruction-level optimizations, have been presented above.

Table 5.2: Profiling for MP3 Implementations

MP3 Code Rev.	1st	2nd	3rd
Original code	Floating Pt. 80.31%	SubBandSynthesis 10.31%	III_stereo 1.43%
Algorithmic Opts.	Floating Pt. 62.73%	III_stereo 6.12%	III_reorder 5.62%
Data & Instruction	SubBandSynthesis 34.32%	inv_mdctL 18.22%	III_stereo 7.32%
Combined Opts.	inv_mdctL 18.98%	III_stereo 8.61%	main 7.87%

We will use the SubBandSynthesis function redesign as a vehicle to illustrate the use of our profiler. In the initial stage, we transferred all critical operations to fixed-point from floating point. The transfer resolved the issue with floating-point operations, but at the same time increased SubBandSynthesis fraction of total energy six times. Next we introduced a series of instruction-level optimizations that resulted in 30% decrease of SubBandSynthesis fraction of total energy, to 34.32% as shown in Table 5.2. In parallel we had decided to try the algorithmic changes on the current code.

Profiling results in Table 5.2 show that the algorithmic optimizations considerably reduced the energy consumption of SubBandSynthesis function - it does not appear in the top three functions, and in fact it is only 3.2% of the total energy consumption. The final step is to combine the algorithmic changes with the data and instruction-level changes, resulting in decrease of SubBandSynthesis fraction of energy consumption to 6% of total.

System and component energy consumptions are shown in Table 5.3 for different revisions of source code optimization. Positive percentages show energy decrease with respect to the original code. Table 5.4 shows the same results, but for performance measurements.

Table 5.3: Energy for MP3 Implementations

MP3 Code Revision	Battery (mWhr)	CPU (mWhr)	Flash (mWhr)	RAM (mWhr)	DC-DC (mWhr)	Lines (mWhr)
Original code	0.446 0%	0.089 0%	0.005 0%	0.178 0%	0.045 0%	0.129 0%
Algorithmic Opts.	0.107 76%	0.020 77%	0.007 -44%	0.040 77%	0.011 76%	0.029 77%
Data & Instruction	0.130 71%	0.025 71%	0.004 27%	0.051 71%	0.013 71%	0.037 71%
Combined Opts.	0.105 77%	0.019 78%	0.007 -41%	0.040 78%	0.010 77%	0.028 78%

Positive percentages show performance increase. Although the energy savings of algorithmic versus data and instruction-level optimizations as compared to original code are comparable, the performance improvement of data and instruction-level optimizations is significant. Note that the increase in energy consumption and the decrease in performance of Flash is due to the increase in code size with the algorithmic change in SubBandSynthesis procedure. The total improvement in system performance and energy consumption more than makes up for the degradation of Flash performance and energy consumption. Combined optimizations give real-time performance for MP3 audio decode which is a primary constraint for this project. In addition, lower energy consumption enables longer battery life. Note that faster implementation that is also more energy efficient might imply higher power consumption, which can be an issue for thermal design of the device. In the case presented in this paper, it was critical to get real-time performance with longer battery lifetime. The average and peak power consumption constraints are met with our final design.

The final MP3 audio decoder compliance to the MPEG standard has been tested as a function of precision for fixed-point computation. We used the compliance test provided by the MPEG standard [38, 39]. The range of RMS error between the samples defines the compliance level. Table 5.5 shows that results. Clearly, the larger number of precision bits results in better compliance. In our final MP3 audio decoder we used 27 bits precision.

Using our design tools to guide software optimization process we have been able to increase performance by 92% while decreasing energy consumption by 77%, with full

Table 5.4: Performance for MP3 Implementations

MP3 Code Revision	System (s)	Flash (s)	RAM (s)
Original code	68.490 0%	0.396 0%	6.309 0%
Algorithmic Opts.	34.562 50%	0.746 -88%	2.776 56%
Data & Instruction	9.185 87%	0.381 4%	4.186 34%
Combined Opts.	5.193 92%	0.718 -81%	2.093 67%

Table 5.5: Fixed-point Precision and Compliance

Precision # bits	Compliance
15	None
20	Partial
27	Full

compliance to the MP3 audio decode standard.

5.5 Processor Specific Code Optimization

The previous section described a general code optimization methodology used to increase energy efficiency of source code. The profiler is used to guide the source code optimization process. As the general code optimization methodology is independent of the system specifications, further optimizations may be possible when the specific characteristics of system components are fully utilized. This section gives an overview of energy efficient optimizations at the source code level that can be utilized for the ARM processor. Energy profiling was done for each code transformation suggested in [3]. An overview of optimizations that have shown significant energy savings follows below. Similar approach can be used to develop source code optimizations aimed at increasing energy efficiency of other processors.

5.5.1 Integer division and modulo operation

The ARM compiler uses shift operation for modulo 2 division since it is much more efficient than the standard division operation. In modulo 2 division unsigned number should be used whenever possible as the unsigned implementation - `div16u` is 14.7% more efficient than the signed version. This is because signed version requires sign extension correction on the shift operation.

```
uint div16u (uint a)    int div16s (int a)
{   return a / 16; }    {  return a / 16;}
```

Whenever possible a condition should be used to replace modulo operation, as it is 51.39% more energy efficient. In example shown below `counter1` implements modulo arithmetic, where `counter2` uses an `if` operator.

```
uint counter1 (uint count)   uint counter2 (uint count)
{                            { if (++count >= 60)
  return (++count % 60);             count = 0;
}                              return (count);    }
```

5.5.2 Conditional Execution

All ARM instructions can be conditionalized. Conditionalizing is done in two steps. First a few compare instructions set the compare codes. Those instructions are then followed by the standard ARM instructions with their flag fields set so that their execution proceeds only if the preset condition is true. Grouped conditions should be used instead of separate `if` statements since they help the compiler conditionalize instructions. In this way 1.25% of energy can be saved. An example of a grouped condition is show below.

```
if (a > 0 && b > 0 && c < 0 && d < 0)
      return a + b + c + d;
```

5.5.3 Boolean Expressions

A more energy efficient way to check if a variable is within some range is to use the ability of the ARM compiler to conditionalize the arithmetic function. An example shown below is 10.6% more efficient than if comparison was done on each coordinate separately.

```
Conditionalized example      Original Code
return ((p.x - r->xmin)      return (p.x >= r->xmin &&
            < r->xmax &&             p.x < r->xmax  &&
        (p.y - r->ymin)              p.y >= r->ymin &&
            < r->ymax);              p.y < r->ymax);
```

5.5.4 Switch Statement vs. Table Lookup

Table lookup is 52.29% more energy efficient than the switch statement when the switch statement codes are more than half of the range of the possible labels. When dense switch statement is used, the table lookup is used to jump to the appropriate case statement. If the case statement contains the call to another function or if it sets a variable, then the table lookup of the address to jump to can be replaced by the code to be executed under the case statement. A good example is shown below where all opcodes are assigned values 0 through 3 thus making the table lookup possible.

```
return  "EQ\0NE\0CS\0CC\0" + 3 * cond;
```

5.5.5 Register Allocation

Usually a compiler cannot assign local variables to a register if their addresses are passed to other functions. If the copy of the variable is made and the address of the copy is used instead, then variable can be placed in the register thus saving memory access. As much as 9.54% energy savings are possible.

If global variables are used, it is beneficial to make a local copy so that they can be assigned to registers. In this way 6.42% of energy can be saved as compared to using a global copy. An example used is shown below.

```
int errs;
void globTest(void)
{   int localerrs = errs;

    localerrs += f2();
    localerrs += g2();
    errs = localerrs;    }
```

When pointer chains are used, it is energy efficient to store a first reference into a variable so multiple memory lookups are not needed. InitPos2 shown below saves 33.9% of energy over InitPos1.

```
void InitPos1(Object *p)   void InitPos2(Object *p)
{                          {   Point3 *pos = p->pos;
    p->pos->x = 0;             pos->x = 0;
    p->pos->y = 0;             pos->y = 0;
    p->pos->z = 0;   }         pos->z = 0;          }
```

5.5.6 Variable Types

The most energy efficient variable type for the ARM processor is integer, it saves 0.39% more energy than short and 18.32% more energy than char. Compiler by default uses 32 bits in each assignment, so when either short of char are used sign or zero extending is needed thus costing at most two extra instructions as compared to ints.

5.5.7 Function Design

By far the largest savings are possible with good function design. Function call overhead on ARM is four cycles. Usually function arguments are passed on the stack, but when there are four or less arguments, they can be passed in registers. A simple example showed over 90% energy savings. Upon return from a function, structures up to four words can be passed through registers to the caller. In this way 72.3% energy can be saved.

When the return from one function calls another, the compiler can convert that call to branch to another function. Energy savings of 49.79% have been observed. An example of such function is shown below.

```
int func1 (int a, int b)
{   if (a > b)
        return (func2(a - b));
   else
        return (func2(b - a));   }
```

Functions that return result that depends only on the value of their arguments and do not have any side-effects can be declared pure. Such functions can then be optimized as common subexpressions by the compiler. Savings of 70% have been shown on a simple example using a square function. Similarly, a functions can be inlined and then no function call overhead is incurred and more optimizations are possible. When square function was inlined we observed 16.89% energy savings. The savings depend highly on the size of the function inlined.

Interprocedural optimization can be done by placing a function definition before its use. An example of that is shown below. Square function is defined before sumsquares, so sumsquares knows what registers square will not use and thus can use those registers for its needs resulting in 24% energy reduction.

```
int square(int x)
{  return x * x;  }

int sumsquares(int x, int y)
{  return square(x) + square(y); }
```

5.5.8 A Complete Example

As a final test of the combined impact of source code optimization, we have manually optimized example code provided by the ARM Inc. [3]. The original source code contained no energy efficient optimizations. Table 5.6 shows that both the general and specific compiler optimizations have a very small effect on the original source code in all categories - the maximum savings are only 0.6%. Once energy efficient source code optimizations are implemented, the savings are much larger – as much as 35% in execution time and 32.3% in energy. Clearly the compiler optimizations make almost no difference in this case as well.

5.6 Summary

I have presented in this chapter a methodology for source code optimization and a tool for profiling energy consumption and performance of software in embedded systems. The profiler is based on the cycle-accurate energy consumption simulator that has been shown

Table 5.6: Complete Example

Energy Opt.	General Opt.	Spec. Opt.	SIZE %change	TIME %change	ENERGY %change
none	Balance	none	0.0	0.0	0.0
none	Time	none	0.0	- 0.2	- 0.2
none	Size	none	0.0	- 0.6	- 0.6
none	Balance	all	0.0	- 0.6	- 0.6
all	Balance	none	-5.8	-35.0	-32.2
all	Balance	all	-5.8	-35.0	-32.3

to give simulation results that are within 5% of hardware measurements. Three major categories of software optimizations have been presented: algorithmic, data and instruction-level. Finally, a set of processor specific optimizations have been discussed that in some cases offer up to 90% of reduction in energy consumption.

I gave an example of application of the software design methodology and the profiling tool to the optimization of MP3 audio decoding for the SmartBadge [51] portable embedded system. The original MP3 source code was obtained from the MPEG standard [37]. Profiling results enabled quick and easy redesign of the MP3 audio decoder software. In addition, I showed the results of evaluating different hardware configurations using our design tools.

The final MP3 audio decoder is fully compliant with the MPEG standard and runs in real time with low energy consumption. Using the design tools and the methodology for source code optimization I have been able to increase performance by 92% while decreasing energy consumption by 77% (see Tables 5.3, 5.4).

Chapter 6

Conclusions

Energy consumption of electronic systems has grown to be of critical importance in the recent years. Both the way systems are designed and the way they are used at run-time significantly affect energy efficiency. In the past, system design was primarily concerned with functionality and performance. Reduction of energy consumption is a relatively new concern that introduces a new trade-off in design of both hardware and software. In this work I introduced new methodologies for energy efficient design of both hardware and software.

Once the system is designed, further energy savings can be obtained with prudent utilization at run-time. As most systems do not need peak performance at all times, it is possible to both transition some system components into low-power states when they are idle (dynamic power management) and to adjust frequency and voltage of operation to the workload (dynamic voltage scaling). In this work I presented two new power management algorithms that enable optimal utilization of hardware at run time. In addition, I developed an optimal dynamic voltage scaling algorithm that saves energy by adjusting processor frequency and voltage according to the load. In the following sections I will summarize the contributions of the thesis and discuss some ideas on what could be done in future.

6.1 Thesis summary

6.1.1 Dynamic Power Management Algorithms

Dynamic power management policies reduce energy consumption by selectively placing components into low-power states. In contrast to heuristic policies, such as timeout, policies based on stochastic models can guarantee optimal results. The quality of results of stochastic DPM policies depends strongly on the assumptions made. In this work I present and implement two different stochastic models for dynamic power management. The measurement results show large power savings.

The first approach requires that only one decision point be present in the system. This model is based on renewal theory. The second approach allows for multiple decision points and is based on Semi-Markov Decision Process (SMDP) model. The basic SMDP model can accurately model only one non-exponential transition occurring with the exponential ones. I presented TISMDP model as the extension to SMDP model in order to describe more than one non-exponential transition occurring at the same time. TISMDP model is very general, but also is more complex. Thus it should be used for systems that have more than one decision point.

Both algorithms show large power savings on four different devices: the laptop and the desktop hard disks, the WLAN card and the SmartBadge. The measurements for the hard disks show that my policy gives as much as 2.4 times lower power consumption as compared to the default Windows timeout policy. In addition, my policy obtains up to 5 times lower power consumption for the wireless card relative to the default policy. The power management results on the SmartBadge show savings of as much as 70% in power consumption. Finally, the comparison of policies obtained for the SmartBadge with renewal model and TISMDP model clearly illustrate that whenever there is more than one decision point available, the TISMDP model should be used as it can utilize the extra degrees of freedom and thus obtain an optimal power management policy.

6.1.2 Dynamic Voltage Scaling Algorithm

I presented a new approach for dynamic voltage scaling that can be used as a part of a power managed system. The dynamic voltage scaling algorithm consists of: (i) change point detection algorithm that detects the change in arrival or decoding rates, and (ii) the frequency setting policy. The policy sets the processor frequency and voltage based on the current arrival and decoding rates while keeping constant performance. I tested my approach on MPEG video and MP3 audio algorithms running on the SmartBadge portable device [51]. The change point detection algorithm is very stable as compared to the exponential moving average algorithm presented previously. As a result, it gives large energy savings at a small performance penalty for both MPEG video and MP3 audio applications. Finally, I implemented the DVS algorithm together with power management algorithms and showed a factor of three savings in energy due to the combined approach.

6.1.3 Energy Efficient Hardware and Software Design

I developed a methodology for cycle-accurate simulation of performance and energy consumption in electronic systems. Accuracy, modularity and ease of integration with the instruction-level simulators widely used in industry make this methodology very applicable to the system hardware and software design exploration. Simulation is found to be within 5% of the hardware measurements for Dhrystone benchmark. I presented MPEG video decoder embedded system design exploration as an example of how this methodology can be used in practice to aid in the selection of the best hardware configuration.

I have also developed a tool for profiling energy consumption of software in embedded systems. Profiling results enable quick and easy redesign of the MP3 audio decoder software. The final MP3 audio decoder is fully compliant with the MPEG standard and runs in real time with low energy consumption. Using my design tools and the software design methodology I presented, performance has been increased by 92% while decreasing energy consumption by 77%.

6.2 Future Work

Although much research has been devoted to energy efficient system design and utilization, this area has not yet reached complete maturity. There are still quite a few limitations to overcome.

The dynamic power management algorithms I presented assume stationary workloads. Although adaptation can be done using the methodology discussed in [14], another approach would be to develop a dynamic scheduler that adaptively changes the mode of operation of system components based on non-stationary workload, thermal control and battery conditions. Such scheduler would need close communication of the energy consumption and performance needs between the operating system, the applications and the hardware. The scheduler would also address the limitation of my dynamic voltage scaling algorithm, namely the need to model the general workload in the active state with the exponential distribution. Clearly better results would be obtained if the workload itself informed the DVS algorithm of its characteristics and future needs. In effect, this approach requires energy-aware operating system that allows the dynamic power manager to have close interaction with the task scheduler and the process manager.

As system designers become more conscious of power dissipation issues and an increasing number of power-optimized commodity components is made available, the new generation of power optimization tools is needed to choose and manage such components, and guide the system designer towards power-efficient implementation. The cycle-accurate energy consumption simulator and profiler are just samples of what might be possible. Similar tools, with many more component models (e.g. model of the wireless link) and multiple abstraction levels are needed. More importantly, the methodology for energy efficient software design is still in its infancy. The compilers are just beginning to consider energy consumption as a criterion in code optimization. The software design methodology I presented is completely manual and would greatly benefit from automation. Some optimizations can be automated at the compiler level, but for others it may be more appropriate to develop a system that can guide the designer in selection and implementation of appropriate optimizations. Energy efficient design and utilization at the system level will continue to be a critical research topic in the next few years as there are still many unsolved

problems and open issues.

Bibliography

[1] Advanced RISC Machines Ltd (ARM), *ARM Software Development Toolkit Version 2.11*, 1996.

[2] Advanced RISC Machines Ltd (ARM), "Fixed Point Arithmetic on the ARM," *Application Note 33*, ARM Inc., September 1996.

[3] Advanced RISC Machines Ltd (ARM), "Writing Efficient C for ARM," *Application Note 34*, ARM Inc., January 1998.

[4] D. Bacon, S. Graham and O. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[5] A. Bavier, A. Montz, L. Peterson, "Predicting MPEG Execution Times," *Proceedings of SIGMETRICS*, pp.131–140, 1998.

[6] L. Benini and G. De Micheli, *Dynamic Power Management: design techniques and CAD tools*, Kluwer, 1997.

[7] L. Benini, G. Paleologo, A. Bogliolo and G. De Micheli, "Policy Optimization for Dynamic Power Management", in *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 6, pp. 813–833, June 1999.

[8] L. Benini, R. Hodgson and P. Siegel, "System-Level Power Estimation and Optimization", *International Symposium on Low Power Electronics and Design*, pp. 173–178, 1998.

[9] M. Berkelaar, *www.cs.sunysb.edu /algorith /implement /lpsolve /implement.shtml*

[10] S. Boyd, *Convex Optimization*, Stanford Class Notes.

[11] Cadence, *www.cadence.com/alta/products*.

[12] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vanduoppelle, *Custom Memory Management Methodology : Exploration of Memory Organisation for Embedded Multimedia System Design*, 1998, Kluwer Academic Pub.

[13] A. Chandrakasan, V. Gutnik, T. Xanthopoulos, "Data Driven Signal Processing: An Approach for Energy Efficient Computing," *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, pp.347–352, 1996.

[14] E. Chung, L. Benini and G. De Micheli, "Dynamic Power Management for non-stationary service requests", *Design, Automation and Test in Europe*, pp. 77–81, 1999.

[15] E. Chung, L. Benini and G. De Micheli, "Dynamic Power Management using Adaptive Learning Trees ", *International Conference on Computer-Aided Design*, 1999.

[16] CoWare, *CoWareN2c url:www.coware.com/n2c.html* .

[17] G. Debnath, K. Debnath, R. Fernando, "The Pentium processor-90/100 microarchitecture and low power circuit design," *International Conference on VLSI Design*, pp. 185–190, 1995.

[18] F. Douglis, P. Krishnan and B. Bershad, "Adaptive Disk Spin-down Policies for Mobile Computers", *Second USENIX Symposium on Mobile and Location-Independent Computing*, pp. 121–137, 1995.

[19] Editors of IEEE 802.11, *IEEE P802.11D5.0 Draft Standard for Wireless LAN*, July, 1996.

[20] A. El Gamal, Z.A. Syed, "A stochastic model for interconnections in custom integrated circuits," *IEEE Transactions on Circuits and Systems*, vol.CAS–28, no.9, pp.888–894, Sept. 1981.

[21] J. Flinn and M. Satyanarayanan, "PowerScope:A Tool for Profiling the Energy Usage of Mobile Applications," *The 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pp.23–30, 1999.

[22] S. Furber, "ARM System Architecture", Addison-Wesley, 1997.

[23] S. Gary, P. Ippolito, "PowerPC 603, a microprocessor for portable computers," *IEEE Design and Test of Computers*, vol. 11, no. 4, pp. 14–23, Win. 1994.

[24] L. Geppert, T. Perry, "Transmeta's magic show," *IEEE Spectrum*, vol. 37, pp.26–33, May 2000.

[25] T. Givargis, F. Vahid, J. Henkel, "Fast Cache and Bus Power Estimation for Parameterized SOC Design," *Design, Automation and Test in Europe Conference*, pp.333–339, 2000.

[26] R Golding, P. Bosch and J. Wilkes, "Idleness is not sloth" *HP Laboratories Technical Report* HPL-96-140, 1996.

[27] K. Govil, E. chan, H. Wasserman, "Comparing algorithms for Dynamic speed-setting of a low-power CPU," *Proceedings of Internactional Conferenc on Mobile Computing and Networking*, Nov. 1995.

[28] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer* vol. 29, no. 12, pp. 84–89, Dec. 1996.

[29] M. Hans and V. Bhaskaran, "A Compliant MPEG-1 Layer II Audio Decoder with 16-bit Arithmetic Operations," *IEEE Signal Processing Letters*, vol. 4, no. 5, May 1997.

[30] M. Hans, "An MPEG Audio Decoder Based on 16-bit Integer Arithmetic and SIMD Usage," *Workshop on Multimedia Signal Processing*, 1997.

[31] D. Helmbold, D. Long and E. Sherrod, "Dynamic Disk Spin-down Technique for Mobile Computing", *IEEE Conference on Mobile Computing*, pp. 130–142, 1996.

[32] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, M. Srivastava, "Power optimization of variable voltage-core based systems," *Proceedings of Design Automation Conference*, pp.176–181, 1998.

[33] I. Hong, M. Potkonjak, M. Srivastava, "On-line Scheduling of Hard Real-time Tasks on Variable Voltage Processor," *Proceedings of International Conference on Computer-Aided Design*, Nov. 1998.

[34] C.-H. Hwang and A. Wu, "A Predictive System Shutdown Method for Energy Saving of Event-Driven Computation", in *International Conference on Computer Aided Design*, pp. 28–32, 1997.

[35] Intel, Microsoft and Toshiba, "Advanced Configuration and Power Interface specification", available at *http://www.intel.com/ial/powermgm/specs.html*, 1996.

[36] T. Ishihara, H. Yasuura, "Voltage Scheduling Problem for dynamically variable voltage processors," *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, pp.197–202, 1998.

[37] ISO/IEC JTC/SC 29/WG 11, "Coded representation of audio, picture, multimedia and hypermedia information," *International Organization for Standardization*, Part 3, May 1993.

[38] ISO/IEC JTC 1/SC 29/WG 11 11172-4, "Information Technology — Coding of moving pictures and associated audio for digital storage media up to 1.5 Mbit/s — Part 4: Compliance Testing," *International Organization for Standardization*, 1995.

[39] ISO/IEC JTC 1/SC 29/WG 11 13818-4, "Information Technology — Generic Coding of Moving Pictures and Associated Audio: Conformance," *International Organization for Standardization*, 1996.

[40] V. Jacobson, C. Leres, S. McCanne, *The "tcpdump" Mannual Page*, Lawrence Berkeley Laboratory.

[41] M. Kandemir, N. Vijaykrishnan, M. Irwin, W. Ye, "Influence of Compiler Optimizations on System Power," *The 27th International Symposium on Computer Architecture*, pp.35–41, 2000.

[42] B. Kapoor, "Low Power Memory Architecutres for Video Applications," *GLS-VLSI*, 1998.

[43] A. Karlin, M. Manesse, L. McGeoch and S. Owicki, "Competitive Randomized Algorithms for Nonuniform Problems", *Algorithmica*, pp. 542–571, 1994.

[44] M. Lajolo, A. Raghunathan, S. Dey, "Efficient Power Co-Estimation Techniques for SOC Design," *Design, Automation and Test in Europe Conference*, pp.27–34, 2000.

[45] S. Lee, T. Sakurai, "Run-time voltage hopping for low-power real-time systems," *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, pp.806–809, 2000.

[46] Y. Li and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems," *DAC*, 1998.

[47] Y. Lu, T. Šimunić and G. De Micheli, "Software Controlled Power Management", *7th International Workshop on Hardware/Software Codesign*, pp. 157–161, 1999.

[48] Y. Lu, E. Chung, T. Šimunić, L. Benini and G. De Micheli, "Quantitative Comparison of PM Algorithms", *Design, Automation and Test in Europe*, pp. 20–26, 2000.

[49] Y. Lu and G. De Micheli, "Adaptive Hard Disk Power Management on Personal Computers", *IEEE Great Lakes Symposium on VLSI*, pp. 50–53, 1999.

[50] Lucent, *IEEE 802.11 WaveLAN PC Card - User's Guide*, p.A-1.

[51] G. Q. Maguire, M. Smith and H. W. Peter Beadle "SmartBadges: a wearable computer and communication system", *6th International Workshop on Hardware/Software Codesign*, 1998.

[52] T. Martin, D. Siewiorek, "The Impact of Battery Capacity and Memory Bandwidth on CPU Speed Setting: A Case Study," *International Symposium on Low Power Electronics and Design*, pp. 200–205, 1999.

[53] H. Mehta, R.M. Owens, M.J. Irvin, R. Chen, D. Ghosh, "Techniques for Low Energy Software," *ISLPED*, 1997.

[54] Mentor Graphics, *www.mentor.com/codesign*.

[55] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[56] B. Noble, M. Satyanarayanan, G. Nguyen, R. Katz, "Trace-based Mobile Network Emulation *Signal and Communications Conference*, 1997.

[57] P. Noll, "MPEG Digital Audio Coding," *IEEE Signal Processing Magazine*, pp. 59–81, September 1997.

[58] OZ Electronics Manufacturing, *PCB Modelling Tools url: www.oem.com.au/manu/pcbmodel.html*.

[59] V. Paxson, S. Floyd, "Wide Area Traffic: The Failure of Poisson Modeling", *IEEE Transactions on Networking*, vol. 3, no. 3, pp. 226–244, June 1995.

[60] M. Pedram, Q. Wu, "Battery-Powered Digital CMOS Design," *Proceedings of DATE*, 1999.

[61] T. Pering, T. Burd, R. Brodersen, "The simulation and evaluation of Dynamic Voltage Scaling Algorithms" *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, 1998.

[62] T. Pering, T. Burd, R. Brodersen, "Voltage scheduling in the IpARM microprocessor system" *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, pp.96–101, 2000.

[63] M. Puterman, *Finite Markov Decision Processes*, John Wiley and Sons, 1994.

[64] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes", *Design Automation Conference*, pp. 555–561, 1999.

[65] Q. Qiu and M. Pedram, "Dynamic power management of Complex Systems Using Generalized Stochastic Petri Nets", *Design Automation Conference*, pp. 352–356, 2000.

[66] D. Ramanathan, R. Gupta, "System Level Online Power Management Algorithms", *Design, Automation and Test in Europe*, pp. 606–611, 2000.

[67] S. Ross, *Stochastic Processes*, Wiley Press, 1996.

[68] Y. Shin, K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," *Proceedings of Design Automation Conference*, pp.134–139, 1999.

[69] T. Simunic, L. Benini, G. De Micheli, "Cycle-Accurate Simulation of Energy Consumption in Embedded Systems," *Design Automation Conference*, pp.867–872, 1999.

[70] T. Simunic, L. Benini, G. De Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems," *International Symposium on Low Power Electronics and Design*, 1999.

[71] T. Simunic, L. Benini and G. De Micheli, "Event-driven power management", *International Symposium on System Synthesis*, pp. 18–23, 1999.

[72] T. Simunic, L. Benini and G. De Micheli, "Power Management of Laptop Hard Disk", *Design, Automation and Test in Europe*, p. 736, 2000.

[73] T. Simunic, L. Benini and G. De Micheli, "Energy Efficient Design of Portable Wireless Devices", *International Symposium on Low Power Electronics and Design*, pp. 49–54, 2000.

[74] T. Simunic, L. Benini and G. De Micheli, "Dynamic Power Management for Portable Systems", *The 6th International Conference on Mobile Computing and Networking*, pp. 22–32, 2000.

[75] M. Srivastava, A. Chandrakasan. R. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Transactions on VLSI Systems*, vol. 4, no. 1, pp. 42–55, March 1996.

[76] Synopsys, *www.synopsys.com/products/hwsw*.

[77] H. Taylor and S. Karlin, *An Introduction to Stochastic Modeling*, Academic Press, 1998.

[78] V. Tiwari, S. Malik, A. Wolfe, M. Lee, "Instruction Level Power Analysis," *Journal of VLSI Signal Processing Systems*, no.1, pp.223–2383, 1996.

[79] V. Tiwari, S. Malik, A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on VLSI Systems*, vol. 2, no.4, pp.437–445, December 1994.

[80] H. Tomyiama, H., T. Ishihara, A. Inoue, H. Yasuura, "Instruction scheduling for power reduction in processor-based system design," *DATE*, 1998.

[81] J. Tong, D. Nagle, R. Rutenbar, "Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic," *IEEE Transactions on VLSI Systems*, vol. 8, no. 3, pp. 273–286, June 2000.

[82] R. Vesilo, "Cumulative Sum Techniques in ATM Traffic Management," *Proceedings of IEEE GLOBECOM*, pp.2970–2976, 1998.

[83] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower," *The 27th International Symposium on Computer Architecture*, pp.24–30, 2000.

[84] M. Wan, Y. Ichikawa, D. Lidsky, J. Rabaey, "An Energy Conscious Methodology for Early Design Exploration of Heterogeneous DSPs," *CICC*, 1998.

[85] M. Weiser, B. Welch, A. Demers, S. Shenker, "Scheduling for reduced CPU energy," *Proceedings of Symposium on Operating Systems Design and Implementation*, pp.13–23, Nov. 1994.

[86] F. Yao, A. Demers, S. Shenker, “A scheduling model for reduced CPU energy,” *IEEE Annual foundations of computer sciend*, pp.374–382, 1995.

[87] “Commercial NiMH Technology Evaluation,” *The 12th Battery Conference*, p.9–15,1997.

[88] “A PSPICE Macromodel for Lithium-Ion Battery Systems,” *The 12th Battery Conference*, p.215–222,1997.

[89] *Workshop on Code generation for Embedded Processors* in *Design Automation for Embedded Systems*, vol. 4, no. 2-3, March 1999.