Dynamic power management techniques for embedded systems

Magesh Kumar Chinnakannan

2009

Magesh Kumar Chinnakannan. (2009). Dynamic power management techniques for embedded systems. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47467>

<https://doi.org/10.32657/10356/47467>

Nanyang Technological University

Downloaded on 26 Apr 2023 20:53:57 SGT

DYNAMIC POWER MANAGEMENT TECHNIQUES FOR EMBEDDED SYSTEMS

MAGESH KUMAR CHINNAKANNAN

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2009

Acknowledgements

When submitting this thesis, I feel thankful for all the help, inspiration and motivation I have received from my supervisor, peers, friends and family. I express my sincere and heartfelt gratitude to my supervisor, **Professor T. Srikanthan**, for his expert guidance, constant support and motivation that were vital for research and development of this thesis. I am very grateful to him for his patience and understanding during the course of this research. I am greatly indebted to him for also providing me financial assistance and a conducive research atmosphere at the Centre for High Performance Embedded Systems (CHiPES). It was indeed a privilege to have the opportunity of working under him.

My sincere thanks to my project mentor **Mohit Sindhwani** who helped and encouraged me throughout the project. I owe a lot to him for his assistance to dispense specialist advice patiently even at the oddest of times. His constructive and creative inputs prompted me to explore new ideas and novel methods during the course of this Masters work. And I would like to thank my fellow researchers **Fauzi, Jagadeesh, R.Mahesh** and **Santanu** who helped me in several ways to protect the best interests of the project. Thanks also to all my other fellow researchers and colleagues, who have made my stay at CHiPES memorable and enjoyable. I extend my thanks to **Ms.Merilyn Yap, Ms.Boh-Nah Kiat Joo** and **Mr.Jeremiah Chua Ngee Tat**, for all the administrative help provided.

Special thanks to the families of **Ravi & Suchitra** and **Balasaravanan & Gayathri** for their unceasing motivation and support. With their unconditional care and affection, they really made me to feel homely during my stay in the University.

I would also like to thank all my friends and role-models, most importantly **AR.Mohan, Arul, David, DG.Senthil, Dipnarayan, Divakar, Ganesh, J.Mohankumar, Rakesh** and **Rajnikanth**. I am greatly indebted to all of them for they not only made my life joyful and exciting, but also be as a source of everlasting support and inspiration.

Finally, I offer my deep gratitude to my parents and master **Sri Nithyananda** for everything that cannot be expressed in words.

Table of Contents

Acknowledgements.....i

Summary.....ix

Chapter 1. Introduction..... 1

 1.1. Motivation..... 1

 1.2. Research Objectives.....2

 1.3. Organization of Thesis.....3

Chapter 2. Literature Survey 5

 2.1. Background.....5

 2.1.1. Embedded Computing Systems5

 2.1.2. Power Management in Embedded Systems.....6

 2.1.3. Dynamic Power Management (DPM) Methodology7

 2.2. System-Level Power Management9

 2.3. Existing Work 11

 2.3.1. CPU-centric Power Management 12

 2.3.2. I/O-centric Power Management..... 14

 2.3.3. Miscellaneous System-Level Power Management techniques:..... 19

 2.4. Miscellaneous Power Management Techniques:.....21

 2.5. Summary & Inferences:22

Chapter 3. Profile-Based Analysis for Dynamic Power Management24

 3.1. Profile-Driven Analysis in Embedded Systems.....24

 3.1.1. Control Flow Analysis26

 3.2. Overview of the Proposed Approach.....26

 3.2.1. Analyzing Control Flow for I/O Power Management26

 3.2.2. Concept of Break-Even Time27

 3.2.3. Analogy: Moving Train in a City30

 3.3. Application Modeling.....31

 3.3.1. CPU and I/O Operations33

 3.3.2. Loops34

 3.3.3. Procedure Calls37

 3.4. Summary38

Chapter 4. Proposed Methodology for Power Management of Peripherals 39

4.1. Overview.....	39
4.2. Algorithm to Analyze CFG for DPM	40
4.2.1. Identification of Switching Points	40
4.2.2. Performance-centric Decision:	41
4.2.3. Profitability-centric Decision:.....	43
4.2.4. Elimination Phase for Redundant Switch-ON points:	46
4.2.5. Handling Loops & Procedure Calls	49
4.3. Simulation Analysis of the Basic Algorithm:	51
4.4. Applications Accessing Multiple I/O Devices:	56
4.4.1. Multiple Sets of Switching Points	56
4.4.2. Simulation Analysis	57
4.5. Peripherals with Multiple Power-Down States:.....	59
4.5.1. Enhanced Algorithm for Multiple Power-Down States.....	59
4.5.2. Simulation Analysis	64
4.6. Evaluation of the Proposed Approach	65
4.6.1. Estimation of Potential Idle Time Durations:	65
4.6.2. Managing Overheads for Efficiency:.....	67
4.6.3. Selection of Significant Switching Points:	68
4.7. Summary	73

Chapter 5. Hardware-Assisted Monitor & Control for DPM 74

5.1. Study of Various Implementation Schemes of the Proposed Approach.....	74
5.1.1. Data obtained from offline analysis:.....	74
5.1.2. Switching Points Causing Effective State-Transitions:	76
5.1.3. Software and Hardware-based Implementation Schemes:	79
5.2. Constructing a Custom Hardware for DPM.....	81
5.2.1. Monitoring the Control Flow of Applications	81
5.2.2. Control Words for Power Transitions.....	82
5.3. Proposed Architecture for the DPM Hardware.....	83
5.3.1. Memory Requirements	84
5.3.2. Monitoring Unit	85
5.3.3. Re-loadable Timer Unit	97

5.3.4. Overall Examination of the DPM-Hardware Operations	99
5.3.5. Qualitative Analysis	101
5.4. Application-Based Scalable Design Schemes	102
5.4.1. Correlation of Switching Point locations	103
5.4.2. Control Word Memory & Timer Design Schemes	104
5.4.3. Selecting a Generic DPM-Hardware Model	107
5.5. Summary	108
Chapter 6. RTOS-Based DPM Framework	110
6.1. RTOS-Controlled DPM Platform	110
6.2. Framework for DPM Utilizing Profile Data	113
6.2.1. Components of the Framework	113
6.3. Performance Analysis	116
6.3.1. Cost Functions of the Overheads:	116
6.3.2. Improving the Efficiency	117
6.4. Summary	120
Chapter 7. Conclusions and Future Work	121
7.1. Conclusion	121
7.2. Scope for Future Work	122
Bibliography	125
Appendix	131

List of Figures

Figure 1.1: Application-centric Dynamic Power Management overview	3
Figure 2.1: Power distribution for a PDA device	6
Figure 2.2: Role of the Power Manager.....	8
Figure 2.3: ACPI Interface and PC Platform.....	11
Figure 2.4: Filter Driver Architecture.....	20
Figure 2.5: Basic Design of PCAP	21
Figure 3.1: Sample section of code represented as a control flow graph	25
Figure 3.2: Break-Even Time Explanation	28
Figure 3.3: Power management from various perspectives	29
Figure 3.4: Weighted CFG representing the Application with Peripheral Accesses.....	33
Figure 3.5: (a) CFG with back edges to the dominators (b) CFG in (a) “ <i>reduced</i> ” to a DAG with only forward edges.....	35
Figure 3.6: (a) Section of CFG having a Natural Loop with single exit (b) Abstracting the loop in with an equivalent ‘Loop Block’.....	36
Figure 3.7: (a) Section of CFG depicting Call to a Sub-routine (SR) (b) Replacing Sub-routine with an equivalent Virtual Block.....	37
Figure 4.1: Flowchart Depicting the Placement of Switch ON “A” points (Step- 1)	42
Figure 4.2: First pass steps of the algorithm.....	42
Figure 4.3: Sample scenario to show the placement of “A” points	43
Figure 4.4: Flowchart Depicting the Placement of “L” points	44
Figure 4.5: Second Pass steps of the algorithm	44
Figure 4.6: Sample scenario showing the placement of “L” points after I/O	45
Figure 4.7: Sample scenario showing the placement of “L” points after “A” points:.....	46
Figure 4.8: Cases of Elimination of “A” when dominated by another “A”	47
Figure 4.9: Cases of Elimination of “A” when dominated by an “I/O”	48
Figure 4.10: Basic steps in the elimination phase.....	48
Figure 4.11: (a) “A” point in a ‘Loop Block’ characterized by Monitoring Count (b) Switching Points utilizing idle periods within a loop.....	49

Figure 4.12: (a) Virtual Block getting selected as Switching Point (b) Shifting the Switching Points to nearer appropriate locations.....	50
Figure 4.13: Model CFG used for Basic Algorithm Simulation.....	52
Figure 4.14: Basic Algorithm Simulation –Final List of switching points.....	55
Figure 4.15: Enhanced Algorithm for Multiple I/O case.....	57
Figure 4.16: Illustrating variable parameters for multiple power-down states.....	59
Figure 4.17: Percentage Energy Savings Vs Sparseness of I/O accesses.....	66
Figure 4.18: Selection of significant switching points	70
Figure 5.1: Assembly Code and Equivalent CFG representation with block addresses	75
Figure 5.2: Basic representation of a switching point data.....	76
Figure 5.3: Traversal Orders of Switching points during program execution ..	77
Figure 5.4: Conceptual View of the DPM Hardware Architecture	84
Figure 5.5: Conceptual View of CAM.....	86
Figure 5.6: Simple RAM and CAM compared.....	87
Figure 5.7: Basic Architecture of a CAM.....	88
Figure 5.8: Block Diagram of a CAM with Important I/O Pins	88
Figure 5.9: Proposed CAM-based Monitoring Unit.....	91
Figure 5.10: Typical Address Space and Example of possible Switching point locations	93
Figure 5.11: Correlation of the Switching Point Locations based on MSBs....	93
Figure 5.12: Bank-Selection Scheme with CAMs.....	95
Figure 5.13: Bank-Selection Scheme with Common-Comparators -Design-3.....	96
Figure 5.14: Timer - Monitoring unit: Interface	97
Figure 5.15: Overall Examination of DPM-Hardware Operations.....	99
Figure 5.16: General Control Word format to control multiple devices.	104
Figure 5.17: Multiple sets of switching points falling at the same location ...	105
Figure 5.18: Control word format with null state	105
Figure 6.1: RTOS-Controlled DPM Platform.....	111
Figure 6.2: Framework for DPM utilizing Profile Data	114
Figure A.1: Basic Algorithm Simulation-CFG with “A” Points Identified...	132
Figure A.2: Basic Algorithm Simulation-CFG with “L” Points Identified	133
Figure A.3: Basic Algorithm Simulation –After elimination phase (final list)	134

Figure A.4: Simulation Analysis for Multiple I/O-1 135

Figure A.5: Simulation Analysis for Multiple I/O-2 (Superimposed with
previous I/O-1 case)..... 136

Figure A.6: Simulation analysis for multiple power-down modes (mode-2). 137

Figure A.7: Simulation analysis for multiple power-down modes (mode-1)-
Superimposed..... 138

List of Tables

Table 4.1: Ordering Switch-OFF points based on their significance 71

Table 4.2: Selection of Switch-OFF points based on their significance..... 71

Table 5.1: Core Signals of a CAM89

Table 5.2: Design examples of FPGA-CAM implementation of various sizes
and performance specifications 90

Table 5.3: Core Signals of the Control Word Memory 92

Table 5.4: Qualitative analysis of core monitoring unit architecture 102

Table 5.5: Choice of architectures based on correlation of addresses 103

Summary

One of the key challenges of Embedded System design is the management and conservation of power. With the increase in user demands for performance and mobility of electronic devices, it results in an equivalent increase in power consumption owing to the complexity of applications. To cope with this increasing power demands, static power reduction techniques such as synthesis of efficient hardware and compilation for low power are applied at design time. In contrast, dynamic techniques leverage on the runtime characteristics to reduce power when systems are serving light workloads or are idle. Dynamic Power Management (DPM) techniques are typically employed at the system-level by adjusting voltage/frequency of processors and subsystems.

This thesis proposes and evaluates a new profile-based methodology to realize dynamic power management of peripheral devices of an embedded application. An algorithm has been developed to facilitate the offline analysis of the control flow information of an embedded application in order to determine profitable regions for energy savings. Extensive simulations have been carried out to demonstrate that the appropriate turn on/off points for the peripherals can be established automatically in order to facilitate proactive control of the peripherals considering both performance and power-efficiency of their operations.

The algorithm has been further extended to cater for applications with multiple independent peripherals and suitable enhancements are incorporated to support devices with multiple power-down modes. Detailed simulation analyses have been carried out to verify that the algorithm is capable of supporting embedded applications with sequential codes, branches, loops and procedures. Hardware assisted implementation schemes to incorporate the dynamic power management of peripherals have been presented. Techniques for customizing the DPM-hardware architecture and its power, delay and area analysis have been presented. Finally, an RTOS-based framework with DPM support has been presented along with suitable techniques to maximize the overall efficiency of the proposed dynamic power management scheme.

Chapter 1. Introduction

This first chapter serves as an introduction to the project and to the thesis report. The first few sections present the motivation and research objectives. Further, a brief outline of the report is presented in the subsequent section.

1.1. Motivation

Most electronic circuits and system designs are confronted with the problem of delivering high performance to cater the increasingly complex applications (e.g. multimedia). This causes an equivalent increase in the power consumption of these devices. As the user demands for mobile systems technology has increased, several complex applications run even on portable devices. Embedded systems being dedicated computing systems employ a combination of hardware and software to perform a specific task. Current day embedded systems drives a multitude of applications with most of them being portable battery-powered devices.

With large color displays and other power-hungry peripherals in an embedded system, power consumption becomes a major concern. Low-power consumption is required to achieve acceptable autonomy for such battery-powered systems. Further, lower power consumption is also required to reduce the operation cost of stationary systems as well as to reduce the environmental impact (e.g. heat dissipation, cooling-induced noise). Hence, achieving highly energy-efficient computation is a major challenge in electronics system design.

Ideally, the problem of managing the energy consumed by electronic systems should be addressed at all levels of design, ranging from low-power circuits, architectures and to application and system software capable of adapting to the available energy source. Many research and industrial efforts are currently underway to develop low-power hardware as well as energy-aware application software in the design of energy-efficient computing systems.

In general, power reduction techniques can be classified as static and dynamic. Static techniques such as synthesis and compilation for low power are applied at design time. In contrast dynamic techniques use runtime behavior to reduce power when systems are serving light workloads or are idle. These techniques are known as Dynamic Power Management (DPM) [1].

DPM techniques are attractive as it finds a wide range of opportunities for system-wide energy savings using software/operating system and also as an alternative to inflexible hardware solutions. However, it presents two major challenges. First, some prediction and estimation about the workloads or the potential idle time durations is required on the part of either the programmer, runtime system, or the compiler to take the DPM decisions. Second, care must be taken that the overheads incurred, such as the additional energy spent in such power transitions do not outweigh the benefits [2] [3].

In light of this and on observing existing techniques for dynamic power management, we identify the need for an application-centric approach for DPM in embedded systems which encompass several peripherals and subsystems. Therefore, in this thesis work, DPM methodologies that are appropriate for an embedded system are investigated and suitable techniques have been proposed and evaluated. In addition, appropriate strategies to implement the proposed techniques so as to integrate the DPM module as part of the overall system are also presented. The following section outlines the research objectives.

1.2. Research Objectives

Current efforts on power management do not often address the application-specific considerations of embedded systems, especially for peripheral operations. Predicting the periods of activity and inactivity (idleness) of peripheral devices based on the application profile can offer key opportunities for energy savings. In order to facilitate effective energy savings during varying workloads, analyzing the characteristics of embedded applications seems to be important and fruitful in emerging design methodologies.

This research project aims to study key application characteristics/profile that can prove vital for predicting the workload of the system. Hence, development of techniques and off-line profiling analysis to efficiently extract salient application-specific parameters as meta-data that can be used for optimizing dynamic power management policies in the system forms the major objective.

With proper, hardware/software modules to communicate to the target platform, this can lead to development of suitable application specific DPM techniques that can be supported by an RTOS (Real-Time Operating System) based embedded system. Figure 1.1 illustrates the overview of such an approach. It is envisaged that application aware strategies for energy centric partitioning and management will lead to efficient DPM solutions for dedicated systems.

Figure 1.1: Application-centric Dynamic Power Management overview

1.3. Organization of Thesis

The remainder of this thesis report is organized as follows:

The following Chapter 2 presents the necessary background and a detailed literature review. This discusses the existing approaches and techniques available for power management and conservation. The limitations in them are highlighted motivating the need for further research.

Chapter 3 presents the concept of application profile-based approach in the context of power management for embedded systems. This chapter serves to explain the key concept behind our proposed approach with a suitable analogy. This chapter also presents in detail the application modeling used in our work.

Chapter 4 explains the proposed methodology for dynamic power management of the peripherals in detail. This chapter explains about the algorithm devised for the offline analysis of an embedded application for key power management decisions. Study of various cases such as applications with single and multiple peripheral accesses and dealing peripherals with multiple power-down modes are presented. Simulation results obtained for each case are presented. Evaluation of the approach and suitable techniques to manage the overheads are also presented.

Chapter 5 presents the study of various possible implementation schemes of the proposed methodology. Investigation on the custom design of a hardware architecture that can serve as a dynamic power manager and evaluation for its suitability are presented along with a qualitative analysis.

Chapter 6 is about the integrated RTOS-based framework we propose for application-centric DPM utilizing the profile analysis data. The performance analysis of the system is analyzed considering the overheads and techniques to improve the overall efficiency are presented. A general flow to realize the framework has also been presented.

Finally, Chapter 7 summarizes the dissertation and briefs on the recommendations for future work. Bibliography and an Appendix showing the simulation analysis results obtained with the control flow graphs under various cases are presented at the end.

The key contributions of the thesis work include the novel profile-based methodology and algorithm for power management of the peripheral devices in embedded systems. Various hardware-assisted implementation designs are presented to incorporate the DPM techniques along with a RTOS-based integrated framework.

Chapter 2. Literature Survey

This chapter presents a detailed survey of existing, relevant methodologies for dynamic power management in embedded computing systems. The chapter begins with the necessary background and introduction. Then the existing power management schemes are reviewed and the limitations of the existing approaches are examined. Further, the motivation and justification for further research especially the need for a DPM scheme that can utilize application profile data are presented.

2.1. Background

The following sections give a brief introduction on embedded systems and the growing concern of higher power consumption and the crucial role of power management in modern design methodologies. This is followed by the description of Dynamic power management (DPM) methodology, fundamental premise and the supporting system models.

2.1.1. Embedded Computing Systems

Embedded systems are dedicated computing systems that are hidden inside other systems and are typically not perceived as general purpose computing systems. Embedded systems are usually intended for a specific application. According to industry experts, more than 98% of the commercial microprocessors manufactured go into Embedded Systems with less than 2% powering PCs [4]. Examples for embedded systems range from washing machines, PDAs and mobile phones to industrial process controllers, automotive control networks and avionics/defense solutions.

Embedded system design attempts to optimize multiple, often contradictory, objectives viz. performance in a real-time constrained environment, cost, robustness, safety & security, time-to-market, etc [4]. The applications of embedded systems are increasing in number, variety as well as complexity. Restricted hardware is used to keep the production costs down in mass-produced items and to keep the weight and power

consumption low in avionics and space applications. Yet, these systems are required to perform complex tasks – for instance, a hand phone uses filtering algorithms to produce a clear sound; PDAs use character recognition algorithms to decode the stylus input on their scratch pads.

2.1.2. Power Management in Embedded Systems

Modern embedded systems have to handle and process data from a wide variety of peripherals. For instance, a modern PDA has Internet connectivity using wireless technologies, digital camera and MP3 players. It becomes important to know where the power in a system is spent and where efforts to reduce power can be best applied. Figure 2.1 is an example power distribution for a PDA class device [5].

Figure 2.1: Power distribution for a PDA device

The power consumption and the corresponding energy consumption of the various components of an embedded system, directly influence battery lifetime, hence the lifetime of the system. Decreased power consumption also results in higher component reliability. Many embedded systems tend to be situated at remote locations; the cost of replacing battery packs is high when the batteries that power these systems fail. Further, the advancements in battery technology are much lower when compared to the growth pace of electronics and embedded systems.

It becomes evident that, a less pervasive system-level issue, but one that is still common, is a need for power management to either minimize heat production or to conserve battery power. As such, the processor, peripherals, software and power supplies need to work together to effectively manage power. It can be recognized that the power management goal in such portable embedded systems is to reduce system-wide energy consumption.

2.1.3. Dynamic Power Management (DPM) Methodology

Dynamic Power Management (DPM) is a design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load on such components. DPM encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited). DPM is used in various forms in most portable (and some stationary) electronic designs; yet its application is sometimes primitive, because its full potentials are still unexplored and because the complexity of interfacing heterogeneous components has limited designers to simple solutions.

The fundamental premise for the applicability of DPM is that systems (and their components) experience non-uniform workloads during operation time. Such an assumption is valid for most systems, both when considered in isolation and when inter-networked. A second assumption of DPM is that it is possible to predict, with a certain degree of confidence, the fluctuations of workload. Workload observation and prediction should not consume significant energy. Dynamic power managers can have different embodiments, according to the level (e.g., component, system, network) where DPM is applied and to the physical realization style (e.g., timer, hard-wired controller, software routine) [2].

Power Manageable Components

The fundamental characteristic of Power manageable components (PMC) is the availability of multiple modes of operation that span the power-performance tradeoff [2]. PMCs are modeled as black boxes. Non-manageable components are designed for

a given performance target and power budget. In contrast, with PMCs it is possible to dynamically switch between high-performance, high-power modes of operation and low-power, low-performance ones.

Another important characteristic of real-life PMCs is that transitions between modes of operation have a cost. In many cases, the cost is in terms of delay, or performance loss. Transition cost depends on PMC implementation: in some cases the cost may be negligible, but, generally, it is not. There might also be a transition power cost: this is often the case when transitions are not instantaneous. Excessive costs may make one or more low-power operation states almost useless, because it is very hard to amortize the cost of transitioning in and out of them. In most practical instances, a PMC can be modeled by a finite-state representation called a power management state machine (PMSM). In general, low-power states have lower performance and larger transition latency than states with higher power.

Power Managed System Model

A power-managed system can be modeled as a set of interacting Power manageable components (PMCs) controlled by a power manager (PM) [2]. Typically, a power manager implements a control procedure based on some observations and/or assumptions of the workload as shown in Figure 2.2. The control procedure is often called *policy*. An example of a simple policy, ubiquitously used for laptops and palmtops, is the *timeout policy*, which shuts down a component after a fixed inactivity time, under the assumption that it is highly likely that a component remains idle if it has been idle for the timeout time.

Figure 2.2: Role of the Power Manager

This simple abstract model holds for many single-chip components like processors and memories as well as for devices such as disk drives, wireless network interfaces and displays that are more heterogeneous and complex than a single chip. Even though the functionality of the Power Manager (PM) is clearly defined, its implementation is not constrained in any way. In some systems, the PM is a hardware block, while in others it is a software routine. Hybrid hardware-software implementations are also possible.

2.2. System-Level Power Management

The previous section presented a general overview of the dynamic power management methodology and the premise for its applicability, underlying assumptions and supporting system models. As mentioned, DPM schemes can be applied at various levels. In this section, we discuss the system-level dynamic power management achieved with the help of software/operating system.

Embedded systems can be viewed as collection of heterogeneous components such as digital VLSI component, optical parts e.g. displays, hard-disk drives (HDD), radio-frequency (RF) components, etc. Such components may be active at different times, and correspondingly consume different fractions of the power budget. To be competitive, an electronic design must be able to deliver peak performance when requested. Nevertheless, peak performance is required only during some time intervals. Similarly, system components are not always required to be in the active state.

Hence, the problem of power management for an embedded system is to reduce the system level power dissipation by shutting of parts of the system when they are not being used and turning them back on when requests have to be serviced. The ability to enable and disable components, as well as of tuning their performance to the workload (e.g., user's requests), is key in achieving energy-efficient designs. Such dynamic power management techniques addressed at system-level can be effectively achieved by software or the operating system and are also referred to as Software-controlled power-management.

The importance of software-controlled power management is described and various existing approaches to manage power consumption in CPU and I/O devices are discussed here with their limitations.

Advantages

There are several reasons for migrating the power manager to software. Software power managers are easy to write and to reconfigure. In most cases, the designer cannot, or does not want to, interfere with and modify the underlying hardware platform. DPM implementations are still a novel art, and experimentation with software is easier than with hardware. Another consideration is that Dynamic power management and the corresponding implementation issues are often considered at the system level. Typical electronic systems are software-programmable, and a majority has an operating system ranging from a simple run-time scheduler or real-time operating system (RTOS) (for embedded applications) to a full-fledged operating system (as in the case of personal computers or workstations). In general, the operating system is the software layer where the dynamic power management policy can be implemented best.

OS-based power management (OSPM) has the advantage that the power/performance dynamic control is performed by the software layer (the OS) that manages the computational, storage and I/O tasks of the system. Implementing OSPM is a hardware/software co-design problem, because the hardware resources need to be interfaced with the OS-based software power manager, and because both the hardware resources and the software application programs need to be designed so that they cooperate with OSPM [2].

Hardware Support

Modern hardware designs provide several features to support DPM. These features include multiple power states in I/O devices and variable-voltage processors. Given these features, a DPM scheme can make decisions on changing the operating voltage of a processor, and switching devices to low-power sleep states during periods of inactivity. As mentioned earlier, states are the various modes of operation that span the tradeoff between performance and power. State transitions have a power and delay cost.

Recent initiatives to handle system-level power management include Microsoft's OnNow initiative and the Advanced Configuration and Power Interface (ACPI) [6] standard proposed by Intel, Microsoft and Toshiba. The former supports the implementation of OSPM and targets the design of personal computers with improved usability through innovative OS design. The latter simplifies the co-design of OSPM by providing an interface standard to control system resources. Figure 2.3 shows the layout of the ACPI standard and Power Management with reference to the Operating System and hardware.

Figure 2.3: ACPI Interface and PC Platform

DPM at the operating system (OS) level has gained importance due to its flexibility and ease of use. The OS, having a global view of system resources and workload, can make more intelligent power-management decisions and can, therefore, provide greater energy savings than hardware-based DPM alone.

2.3. Existing Work

The existing work reported in the literature encompassing a range of approaches and techniques for software-controlled power management are presented with their advantages and limitations. Here, the approaches are broadly categorized as CPU-centric, I/O device- centric and miscellaneous system-level power management schemes.

2.3.1. CPU-centric Power Management

With the recent IC technology advancements, many modern processors can vary its voltage and working frequencies (and thus processing speeds) dynamically. Dynamic Speed Scaling or voltage scheduling enables considerable CPU energy savings. These techniques make use of the fact that energy consumption of CPU varies as the voltage/frequency changes. Considerable efforts have been put in the past to come up with various optimal scheduling algorithms depending upon the nature of the hardware, applications, etc.

Voltage Scheduling

Energy consumption of CMOS circuits has a quadratic dependence on the supply voltage. Lowering the supply voltage is the most effective way of reducing energy consumption. However, lowering the supply voltage also decreases the clock speed. The dynamic power consumption of CMOS circuit is represented by the following equation:

$$P_{\text{dynamic}} = C_{\text{eff}} \cdot V_{\text{dd}}^2 \cdot f_{\text{clk}}$$

C_{eff} is the effective load capacitance, V_{dd} is the supply voltage and f_{clk} is the clock frequency.

This trade-off introduced various voltage-scheduling techniques.

Static and dynamic voltage scheduling have been proposed with the latter drawing more attention (DVS). DVS techniques change the clock speed and its corresponding supply voltage dynamically, to the lowest possible level, while meeting the task's deadline constraint under the control of Software.

In real time systems, the slack time and variations in workload from time to time can be exploited to lower the supply voltage. There are two types of DVS approaches depending on the scaling granularity [7]. Intertask dynamic voltage scheduling (InterDVS) determines the supply voltage on task-by-task basis, while Intratask dynamic voltage scheduling adjusts the supply voltage within an individual task boundary. With multiple tasks, the InterDVS techniques assign the proper speed to each task dynamically while guaranteeing all their deadlines.

InterDVS has several practical limitations. For instance, since a task scheduler in OS determines the supply voltage of a task, it requires OS modifications. Furthermore, it cannot be applied to a single task environment, because the supply voltage is determined as a constant value for a given task. For many small-sized embedded mobile applications that are based on a single task model, this can be detrimental to a wide adoption of variable voltage processors in practice. Even in a multi-task environment, InterDVS may not be effective in energy reduction if the execution time of one task is dominant in the total execution time [8] [9] .

IntraDVS serves as an effective technique for developing low power single-task applications or low power multitask applications, where a small number of tasks are dominant in total execution time. It has an advantage that existing OS can be used without any modifications. The main issue of IntraDVS algorithm is to select the program points where the voltage and clock will be scaled.

Segment-based IntraDVS techniques partition a task into fixed-length segments. After executing a segment, it adjusts the clock speed and supply voltage exploiting the slack times from the executed segments of a program. Path-based IntraDVS use the control flow information to find slack times. Selecting all the program locations where the changes of remaining workload can be identified, it inserts voltage scaling codes in compile time. This code is executed at run-time exploiting all slack times coming from run-time variations of different execution paths. Predicting the remaining execution cycles effectively and determining the voltage-scaling points are key issues in Path-based IntraDVS. Techniques such as using the remaining worst case execution path, remaining average case execution path, optimal-case execution path have been proposed in the literature [7] [9].

IntraDVS is also achieved utilizing the stochastic information of a program's execution time. Another technique referred as Memory-aware IntraDVS utilizes the CPU idle times due to external memory stalls. Compiler-driven and event-driven approaches have been proposed.

An improved approach by Nevine et al. [10] involves collaborative efforts of Operating System and compiler for voltage scaling. In the offline stage, the compiler helps in annotating an application's source code with path-dependent information. OS

periodically changes the processor's frequency and voltage (DVS) based on the temporal information.

Miscellaneous techniques for CPU Power Management:

Register files provide a good potential for controlling power/performance profile since their use is often directly controlled by the compiler and these constitute an architectural feature present in virtually all types of processors. Their role in operation delay, numbers being used, data width and the number of ports impact power consumption. In [11] the compiler generates configuration code embedded in the application and produces different code versions to be selected at run-time to adapt the code to be run on different architectural organizations. Instruction scheduling algorithms and dynamic reconfiguration of hardware resources according to the parallelism in the program using runtime profiling techniques are some of the other approaches reported in the literature.

2.3.2. I/O-centric Power Management

While reducing CPU power results in significant energy savings, the I/O subsystem is also a potential candidate to target for power reduction in systems that are I/O intensive. Most embedded systems tend to be I/O intensive and many of them require real time guarantees. Most I/O-based DPM techniques save energy by resource shutdown. That is, identifying time intervals where I/O devices are not being used and switching these devices to low-power modes during these periods. However, switching between device power states has associated time and power penalties. As this deals with putting the peripherals in low power states, it is also referred as Dynamic Power Down [12].

Power management using predefined set of policies, mathematical scheduling of I/O operation for effective utilization of idle times and other miscellaneous techniques are discussed below. Compared with CPU-centric power management, relatively fewer techniques are reported in the literature for I/O-centric power management.

Power Management Policies:

Regarding this I/O-centric DPM, the rules to determine whether to shutdown a device are called policies [3]. These power management policies can be classified into three categories based on the methods to predict the justifiable idle times. These categories are Time-out, Predictive and Stochastic.

Time-out Policies:

Time-out based DPM schemes shut down I/O devices when they have been idle for a specified threshold interval. The next request generated by a task for a device that has been shut down wakes it up. The device then proceeds to service the request. Since the early 1990s, manufacturers have been recommending spinning down hard disks after some period of idleness, which is a simple example of the time-out policy [13] [14].

A time-out policy has a time-out value τ . Time-out policies assume that after a device is idle for τ , it will remain idle for at least a minimum duration of time called Break-even time T_{BE} . Simply defined, the minimum length of an idle period to save power is called the break-even time [2]. An obvious drawback is the energy wasted during this time-out period. Time-out-based policies include fixed-timeout, such as setting τ to three minutes. This is widely adopted in commercial products. In Microsoft Windows, users can set the time-out values for hard disks and monitors in the control panel.

Alternatively, time-out values can be adjusted at runtime [15]. For example, one adaptive time-out policy (ATO) adjusts τ by the ratio of current τ and the previous idle period. If the ratio is too small, this policy increases τ ; if the ratio is too large, it decreases τ . Some time-out policies consider the variations of hardware parameters. These device-dependent time-out policies select τ based on the break-even time of the device under control.

Predictive Policies:

Predictive policies predict the length of an idle period before it starts, eliminating the time-out period [16]. If an idle period is predicted to be longer than the break-even time, the device sleeps right after it is idle. Requests make a device change between busy and idle. The lengths of busy and idle periods form two sequences: $B[1], B[2], \dots$ and $I[1], I[2], \dots$. Let $B[n]$ be the busy period before idle period $I[n]$ for any integer n .

One study finds that busy and idle periods form an “L” shape [16]. Short busy periods are followed by long idle periods; long busy periods are followed by short idle periods. Therefore, the L-shape policy shuts down a device if it is busy for only a short period of time. However, this policy does not handle the situation when short busy periods are followed by short idle periods near the origin.

Few other predictive policies use only past idle periods to predict the length of future idle periods. Adaptive learning tree (LT) encodes the sequence of idle periods into tree nodes [17]. This policy predicts the length of an idle period with finite-state machines similar to multi-bit branch prediction in microprocessors. If an idle period is predicted longer than T_{BE} and it is indeed longer than T_{BE} , the confidence level increases; otherwise, the confidence level decreases. This policy uses the history encoded on the nodes and the confidence level to compute a node traversal path and to determine the power states for future idle periods.

Another predictive policy uses the predicted and the actual lengths of a previous idle period [18]. Suppose $\rho[n]$ is the prediction for $I[n]$, then

$\rho[n+1] = a \times I[n] + (1 - a) \times \rho[n]$; here a is a constant between zero and one.

This policy is called exponential average (EA) because $\rho[n+1]$ essentially is the average of $I[1], [2], \dots, I[n]$ weighted by an exponential sequence. This policy also limits the speed of growth for ρ , $\rho[n+1] \leq c \times \rho[n]$ for a constant $c > 1$, so that the prediction is not dominated by one exceptionally long idle period.

Stochastic Control:

Stochastic policies model the arrival of requests and device power-state changes as stochastic processes, such as Markov processes. Minimizing power consumption is a stochastic optimization problem [19]. A simple stochastic policy models requests and power-state transitions as stationary discrete-time Markov (DM) processes. At any given time, a request arrives with a probability; the device sleeps with another probability. The optimal probability for the device to sleep can be obtained by solving the stochastic optimization problem. Such a solution is valid only for geometrically distributed stationary stochastic processes.

There are two ways to generalize this model: extending it to other stochastic models and including non-stationary behavior. The first extension can be achieved by using continuous-time Markov models or continuous-time semi-Markov models [20] [21]. With these models, there is no need to evaluate the appropriate power states periodically. Instead, the arrival and service of requests are the events that trigger state transition decisions. Eliminating periodic state evaluation (as done in DM) reduces processor power consumption. Moreover, requests and devices can be modeled by stochastic processes that are not necessarily geometrically or exponentially distributed; thus Markov processes cannot capture them. For example, a study finds that the requests for wireless communication are best modeled by Pareto distributions [21]. This study proposes the use of time-index semi-Markov (SM) models and determines a way to compute optimal policies.

The second extension considers non-stationary (NS) requests [22]. One approach computes the optimal policies in advance for different sets of arrival rates and stores the results. The actual rates are estimated at runtime. If the estimated rate is the same as any pre-computed rate, then the pre-computed optimal policy is used. Otherwise, interpolation is applied between these pre-computed solutions to obtain the sleeping probability.

I/O Device Scheduling

Though the above mentioned I/O DPM methods provide significant energy savings in non-real time systems, they are not viable candidates for use in real time systems. Due to their inherently probabilistic nature, these methods lack the important aspect of providing real-time temporal guarantees. The critical nature of I/O devices operating in real-time prohibits the shutting down of peripherals during run-time in order to avoid the catastrophic consequences of missed task deadlines. Algorithms that operate in a deterministic manner are needed in order to meet real-time behavior.

D.Li et al.[23] rely on the notion of a mode dependency graph for multiple processors and I/O devices. Valid mode combinations are generated using topological sorting. Then a sequence of modes is determined for each resource such that all timing constraints are met and max-power requirements are satisfied for a given task-set. A schedule generated here is not necessarily an energy optimal schedule for the task set.

Swaminathan et al. [24] have worked on energy-conscious, deterministic I/O Device Scheduling in hard Real-Time Systems. Differing from [23] which doesn't distinguish between I/O devices and processors, their work is that of a set of periodic tasks executing on a single processor. They present two online device scheduling algorithms for hard real-time systems without missing task deadlines. An advantage of online I/O device scheduling is that online DPM decision making can further exploit underlying hardware features such as buffered reads and writes. They had inputs as a set of predetermined task schedule and a device-usage list for each task. Their algorithm called LEDES (Low Energy Device Scheduler) guarantees that real-time constraints are not violated. Another version of this algorithm for handling I/O devices with multiple power states is termed as multi-state constrained low-energy scheduler (MUSCLES).

But in this work, only non-preemptive task scheduling is supported. The same authors came up with an energy-aware algorithm for preemptive schedules termed Maximum Device Overlap which is an offline method. A deficiency in this approach is that it is hard to adapt to changes in the operating environment, such as dynamic task join/leave or early job completion. Hui Cheng and Steve Goddard in [25] propose an online Energy-Efficient Device Scheduling (EEDS), for hard-real time systems based on the preemptive periodic task model. This algorithm uses preemptive Earliest Deadline First to schedule jobs. As compared with [24], EEDS performs inter-task device scheduling rather than intra-task device scheduling. That is, the scheduler does not put devices to sleep while tasks that require them are being executed, even though there are no pending I/O requests at that time.

Linwei Niu and Gang Quan in [12], present a dynamic scheduling algorithm to minimize the energy consumption by both the DVS processor and peripheral devices in a weakly hard real-time system. In this, they first use a static approach to partition real-time jobs into mandatory and optional part to meet the weakly hard real-time constraints. Then an on-line approach is adopted which can effectively exploit the run-time variations and reduce the preemption impacts to leverage the energy saving performance. This approach doesn't directly deal with I/O device scheduling. It is more of a peripheral-conscious processor voltage scheduling rather than a CPU-alone

DVS techniques. They attempt to employ both DVS and Dynamic Power Down judiciously to save system-wide energy for weakly hard-real time systems.

2.3.3. Miscellaneous System-Level Power Management techniques:

The OS is most effective in power managing components that are explicitly controlled through device drivers. **Workload monitoring** is important to achieve effective OS directed dynamic power management. One of the earliest works [26] presents a minimally intrusive monitoring tool to collect information that could be used to drive the policy's decision and make it available to the power manager for use. Information regarding the usage of each hardware resource, such as, the distribution of inter-arrival times of request to the resources and the distribution of service times for the requests are extracted from a stream of time-stamped events.

A software routine is used to perform this update. The monitoring is done by calling this routine before the driver sends execution commands to the hardware and immediately after the driver becomes ready to communicate to the kernel that the hardware has completed the required service. This approach requires simple modification of the device driver with the insertion of the call to this routine. In this approach, the CPU is monitored by detecting the scheduling of the idle process.

Y.H.Lu et al. present a software architecture in [27], which exploits the Advanced Configuration and Power Interface (ACPI) concept enabling investigation of power management algorithms in a systematic fashion. The template is implemented as kernel-level **filter drivers (FD)** that attach to the device drivers from hardware vendors. The FD also reports device utilization for future management decisions. This is shown in the Figure 2.4.

The power manager (PM) is a separate program that collects device utilization information from the FD and issues state-transition commands to it which in turn commands the device. The dashed line shows that the PM may obtain information from applications regarding future requests, which is possible only if the application supports this strategy. As the PM resides outside the OS, it makes designers to evaluate various DPM algorithms on real- machines without considering many details.

Figure 2.4: Filter Driver Architecture

Most of the DPM policies do not distinguish request sources. Considering the fact that different processes consume different amounts of power, **Process-based power management** with a system level model of requesters has been proposed in [28]. This approach uses kernel information for estimating device utilization and shutting down idle devices. A matrix is constructed which has details about the utilization of each devices against every requesting processes. Termed as Device-requester utilization matrix U , each element $U(d,r)$ represents the utilization of device d by requester r . Similarly a vector P is constructed for processor utilization details. The key decision made by the power manager is to shut down a device d if its total utilization falls below a particular threshold value [29].

In [30], **Performance-based Application Programming Interface** is used to enable processes to specify their device requirements before generating requests. Requesters actively specify which devices will be used through an API-based function call: *RequireDevice* (*device, type, period, wait*). The four parameters used by the function represent the device used, type of usage, period of usage and the waiting time respectively. Process scheduler adjusts execution orders based on requests reported from Device drivers and performance requirements from requesters. Then the lengths of idle periods reported to power manager.

Program Counter-based prediction techniques has been proposed in [31] to predict I/O activities in the Operating system. Correlating I/O operations The *Program Counter Access Predictor (PCAP)* dynamically learns the access patterns of applications and predicts when an I/O device can be switched to a low-power mode to save energy. It uses path based correlation to observe a particular sequence of program counters leading to each idle period and predicts future occurrences of that idle period. This concept is motivated by branch prediction in high performance microprocessors. If a long idle period is detected the program counters following the last I/O operation are recorded to be able to identify future occurrences of this program phase before the idle interval starts.

Figure 2.5: Basic Design of PCAP

To implement efficiently, the path is encoded as *path signature* resulting in fewer bytes variable. This along with the look-up prediction table is depicted in the Figure 2.5.

2.4. Miscellaneous Power Management Techniques:

Application-Supported Power Management

Power management can be effectively done with suitable support from the applications [32] [33]. Since benefits from the applications adaptations or programmer support is not feasible from an embedded system perspective, only those approaches that are applicable to embedded systems are discussed here. Some other application controlled power management though has much better potential for reducing energy consumption; it places the burden of inserting power management directives on the programmers and requires the existing applications be modified.

Code-transformations (explicit I/O based applications) that increase device idle times is the key idea proposed by Heath et al. in [34]. After clustering disk read operations (taking advantage of buffer-space and acceptable delay-threshold), the operating system is informed about the length of each upcoming period of idleness-through a system call. A compiler framework (based on SUIF2) for performing the transformations automatically has been devised.

Increasing the expected idle periods using some external hardware is yet another approach for power management. Using buffers to extend idle periods, with buffer size being a function of network bandwidth, using dedicated caches to produce idle periods between memory operations have also been reported as a way to achieve power management especially for high energy-delay efficiency, streaming media applications [35]. Other similar, miscellaneous power management schemes have been outlined in [36]-[38].

2.5. Summary & Inferences:

Power consumption is an important design issue in embedded systems. Battery lifetime and system lifetime can be extended by reducing energy consumption in an embedded system. Dynamic power management is a powerful methodology for reducing power consumption at system-level with the help of software circumventing inflexible hardware solutions. In a power-managed system, the state of operation of various components is dynamically adapted to the required performance level, in an effort to minimize the power wasted by idle or underutilized components. For most system components, state transitions have non-negligible power and performance costs. Thus, the problem of designing power management policies that minimize power under performance constraints is a challenging one. Several power management policies and techniques were surveyed. Even though dynamic power management has been successfully employed in many real-life systems, much work is required for achieving a deep understanding on how to design systems that can be optimally power managed.

It is observed that relatively fewer techniques and possibilities are explored in the literature for managing the power consumption of I/O subsystem compared with that

of CPU power management techniques. Another key observation is I/O-centric approaches do not appear to incorporate the application-specific considerations of real-time and embedded systems. Even though some of the CPU-centric techniques accounts for profile-driven power management such as in [8]-[10], the feasibility of such techniques are yet to be explored for power management of I/O subsystem. The implementation issues faced in PC and embedded platforms and the importance of Operating system support are also evident. Another observation is that the ACPI (Advanced Configuration and Power Interface) concept which helps in abstracting the platform's (hardware) power management features into a standardized interface and allow for standardized power management policies, seems to be prevalent in PC platforms but yet to be optimized for real-time and embedded systems.

As the efficiency of DPM policies typically depend on the dynamic characteristics of an application, suitable off-line techniques need to be devised for the efficient identification of the salient parameters of the proposed policy. From an embedded system perspective, predictive techniques to determine the periods of activity and inactivity (idleness) of peripheral devices based on the application profile can offer key opportunities for energy savings. Further, care must be taken that the overheads incurred, which include increased delay in the wake-up of the device when it is needed and the additional energy spent in such power transitions do not outweigh the benefits.

As such, we investigate the significance of a profile-based approach for DPM in the next chapter and then propose suitable techniques to efficiently extract salient parameters and to utilize this in a runtime system in the subsequent chapters.

Chapter 3. Profile-Based Analysis for Dynamic Power Management

This chapter highlights the importance of a profile-based analysis in embedded systems in general and addresses the issue of power management utilizing such techniques. This chapter serves as the background to present our proposed methodology. As such, this presents an analogy to our approach and the application modeling of our work.

As examined in the previous chapter on literature survey, existing methods for the power management of I/O subsystem have not often addressed application-aware strategies. It is evident that a profile-based approach will lead to efficient DPM solutions for dedicated systems. The application profile can be studied to identify the points of interaction with the peripherals and the inter-arrival times between successive requests for the peripheral access. In this chapter, the importance and significance of profile-driven approach in embedded systems, in general and for power management in particular is presented. As we propose a methodology in the next chapter, to identify the idle time durations between successive I/O service requests, the overview of the proposed approach is presented with a suitable analogy. Before elaborating on the proposed methodology in subsequent chapter, this chapter discusses on modeling of the application used in our work.

3.1. Profile-Driven Analysis in Embedded Systems

Identifying key characteristics of an application is essential from an embedded system design perspective. It is one of the most used techniques in code/compiler optimization, hardware/software co-design, etc. which aim for more power/area/time-efficiency [39] [40]. Program development, maintenance and resource usage can be greatly improved by the static and dynamic analyses that provide information about application behavior [41].

The Control Flow Graph (CFG) is a fundamental data structure needed by almost all static and dynamic analyses techniques. Such analysis includes global data-flow analysis, the construction of static single assignment graph and data-dependence analysis [42]. The CFG which is a representation through basic blocks and their connecting edges enables to know the flow between the blocks thereby providing an idea about the possible flows of execution for a program [43][44]. There are various commercial and open-source tools to generate CFGs of the source code by performing static analysis with certain safe assumptions [39][40][45]. Having the knowledge of the possible inputs and most critical and/or frequent paths in the design, these assumptions can be refined further. Figure 3.1 illustrates a sample representation of a control flow graph for a simple code.

Figure 3.1: Sample section of code represented as a control flow graph

3.1.1. Control Flow Analysis

The importance and significance of a control flow profile-based analysis is much appreciated and very well utilized in several significant areas of embedded systems research. One such important example which relies heavily on control flow information is the Worst Case Execution Time (WCET) analysis used in hard-real time system design [46] [47]. To derive the execution time of a program, both the characteristics of the program code and the computer hardware must be considered. These techniques are based on certain assumptions like: One Specific program running in isolation, running on a certain CPU & Clock, compiled with a certain compiler, no interfering background activity and no task switches.

Further as discussed in literature review, in dynamic power management also the use of a control flow graph is very much essential for scaling the voltages of the CPU at various locations of the program [8] -[10]. In addition to these techniques, few other closely related techniques that rely on the control flow graph for power management and conservation are reported in [48] and [49]. Almost all of these techniques deal either with CPU voltage scaling or power management of micro-architectural functional units.

3.2. Overview of the Proposed Approach

This section presents an overview of the proposed approach by outlining the central idea to address profile-based power management of the peripherals. An explanation about *Break-even time* - a concept used in dynamic power management is presented which is an important device feature. Then the proposed methodology is illustrated with a suitable analogy.

3.2.1. Analyzing Control Flow for I/O Power Management

Our approach attempts to utilize the control flow profile of an embedded code to take power management decisions of the peripherals rather than the CPU, with due considerations given to both performance and power. We focus on static analysis techniques assuming uninterrupted program execution on a single processor. A Control flow graph (CFG) mimics an embedded application where certain nodes of the

graph represent CPU-related operations and others represent I/O device related operations.

From such an application profile, given the estimates of least possible execution times of the nodes of the CFG, one can estimate the possible idle-time durations between successive I/O requests. Such control flow based timing information gives a predictable knowledge about the operations of the devices. Given the peripheral power consumption parameters, suitable decisions can be taken for both performance-centric functioning of the peripherals as well as to put them in low power mode for profitable energy savings. Focusing on this concept, a control-flow profile based approach is proposed which is outlined as follows:

1. The CFG representing an embedded application needs to be analyzed offline using suitable algorithms that can estimate the potential idle time durations. An I/O device when it is required of its service needs to be turned ON at appropriate moment without neglecting performance. Hence suitable locations in the application have to be chosen for switching it ON. Similarly, when potential idle time durations are detected, the device can be turned OFF for profitable energy savings. Such locations in the application are appropriate for switching it OFF.
2. Finding those suitable locations in the CFG for both performance-centric functioning as well as profitable energy savings. Thus, the algorithm has to determine the placement of *Switching Points* at suitable locations. The basic inputs that are needed are the estimates of least possible execution times of basic blocks in a CFG (at any abstraction level) and peripheral parameters such as Turn-ON time and *Break-even time*.
3. Incorporation of the algorithm for scenarios where application can possibly deal with single peripheral access, multiple peripheral accesses and peripherals with multiple power-down states.

3.2.2. Concept of Break-Even Time

An important device feature used in dynamic power management is Break-even time. Power management is a prediction problem; it seeks to forecast whether an idle period

will be long enough to compensate for the overhead of power state changes. The minimum length of an idle period to save power is called the *Break-even time* (T_{BE}) [3].

It depends on individual devices and is independent of requests. Let a device have a state transition delay of T_o (including shutdown and wake-up delays), transition energy of E_o and its power consumption in the working and sleeping states be P_w and P_s . In Figure 3.2, the areas A and B represent the energy spent during shut-down and wake-up of the device. To compensate for the energy consumed in changing states, the device must stay in the sleeping state at least for duration of T_{MS} termed as the minimum sleeping time.

Figure 3.2: Break-Even Time Explanation

In short, the Break-even time makes energy consumption in both cases equal.

$$P_w \times T_{BE} = E_o + P_s \times (T_{BE} - T_o) \quad (3.1)$$

The break-even time has to be larger than the transition delay. So,

$$T_{BE} = \max [(E_o - P_s \times T_o) / (P_w - P_s), T_o] \quad (3.2)$$

$$T_{BE} = T_{OFF} + T_{MS} + T_{ON} \quad (3.3)$$

The sum of device turn ON time, required minimum sleeping time (when applicable) and device turn OFF time is termed as the Break-even time.

Figure 3.3 illustrates the concept of power management and break-even from the viewpoints of workload, device and power states. A workload consists of multiple requests. For hard disks, requests are read or write commands; for network cards, requests are packets to send or to be received.

Figure 3.3: Power management from various perspectives

When there are requests, the device is busy; otherwise, it is idle. Here, the device is idle between T_1 and T_4 . When the device is idle, it can be shut down to enter a low-power sleeping state. In this illustration, the device is shut down at T_2 and woken up at T_4 , when requests arrive again. Changing power states takes time; T_{OFF} and T_{ON} are the shutdown and wake-up delays. In the example of hard disks and displays, it takes several seconds to wake up these devices.

Furthermore, waking up a sleeping device may take extra energy. In other words, changing power states has overhead. If there were no overhead, power management would be trivial: Just shut down a device whenever it is idle. Unfortunately, there is delay and/or energy overhead. Consequently, a device should sleep only if the saved energy justifies the overhead. In Figure 3.3, the device does not sleep immediately after it is idle. Its duration in the sleeping state is more important than the length of this idle period. Power management policies predict whether $T_4 - T_3 > T_{MS}$ instead of whether $T_4 - T_1 > T_{BE}$.

3.2.3. Analogy: Moving Train in a City

A simple analogy illustrates the proposed idea. Consider a single train traveling in a city. Assume there are several possible tracks between the starting station and the destination with possible intersections among them. Further, there will be regions where the train may have to travel intersecting busy business districts in the city. When the train is traveling in such a region of interest, the whole traffic in the city has to come to halt. This causes a waste of the resources as far as the business in the district is considered. This can be tolerated during times when the train is actually crossing that area, i.e., the train really needs the service of that track. But it will not be justifiable if people in those areas need to wait for unknown duration of time waiting for the train to let go. This scenario will be worsened further, if there is insufficient time for the users to cross before another train comes along the track.

To tackle the situation, the information we need is both the behavior of the traveling patterns of the train and its possible time of arrival at each station. Then, a method can be devised to enable the people to use the business-area track for a certain duration estimating the possible next arrival of the train along that track.

With that analogy, the control flow based approach for peripheral power management can be comprehended as follows. The train moving along several possible tracks is similar to the program execution which can take different paths of control flow of the application. The peripheral accesses in the CFG are the usage of business districts in the city. The problem is to limit the usage of peripheral accesses only when they are needed. Further, by estimating the time for next possible request of the peripheral service, the device can be turned off, thereby saving the energy resource. Again, by predicting its next usage, the device can be turned ON for using its service.

As power management does not change requests, we are concerned about only the power when a device is in either the working or sleeping state. We don't consider the power required to serve requests.

3.3. Application Modeling

In this work, the Control Flow Graph (CFG) data structure with possible edges & nodes represents the embedded application and the choice of abstraction level at which it is obtained may differ. For example, the CFG can be generated either at the source code level or at the assembly code level. The difference between the two comes from the compile optimizations. The nodes of a control flow graph (CFG) correspond to basic blocks, and the edges correspond to control transfer between basic blocks. We consider the general form of an embedded application that may consist of sequential code, branches, loops and procedures. In this document, the terms ‘nodes’ and ‘blocks’ are used interchangeably.

Furthermore, we require that the CFG supplied as the input is *safe*, i.e., a safe approximation of a control flow graph (CFG) does not miss any feasible paths of the application. Such a CFG approximation is an upper bound of the set of nodes and edges representing the control flow of the application program.

The following is the principal theoretical structure of a program:

1. A program contains routines,
2. Routines contain basic blocks,
3. Basic blocks contain instructions which gets executed in sequence,
4. Instructions contain operations,
5. The edges between blocks may be intraprocedural edges and/or interprocedural edges.

It can be noted that a call instruction in the machine code might be a good hint for an interprocedural edge and a jump might be a good hint for an intraprocedural edge, but in the conceptual view, a call to a routine may be implemented by both instructions, and even by other means, and in the same way, an intraprocedural edge may be implemented with a call. Here we assume that an edge never returns, i. e., if an edge is traversed twice, no information about the first traversal of the edge is used in the program. Intuition suggests that a stack structure should not be used inside routines, but only between routines, to determine control flow (e.g., for return addresses).

Furthermore, similar to the Worst-Case Execution Time (WCET) analysis, the program being analyzed should be free from unbounded loops, recursion and dynamic function calls.

Complexity of CFG construction:

As the CFG is used as the key component for analysis, a brief explanation about the complexity of CFG construction from an application code is presented here from literature [42]. Among several techniques used, [42] was chosen as it presents clear details for CFG construction from assembly code. Their algorithm has three distinct steps: detecting and marking labels, adding standard control flow, and adding control flow that originates in delay slots (i.e. a delayed branch target which has to be resolved).

The first two steps constitute the standard CFG-construction algorithm. They take time that grows linearly with the program's length. If there are no branches in delay slots, they construct a valid CFG. When branches occur in delay slots, the third step is needed to model the program's behavior and construct the corresponding control flow. This step adds control flow that results from branches in delay slots that are ignored by the previous steps.

Let 'i' be the number of instructions, 'b' the number of branches and 'j' the number of potential branch targets. The first step examines each instruction once and thus, it takes $O(i)$ times. The second step, for a branch, however, it must add j edges, making the time for the second step is $O(i + j \cdot b)$ which in the worst-case becomes $O(i^2)$.

In the third step, as control passes from one block to another, the algorithm passes to the target blocks a list of pending control-flow instructions with a countdown timer for each that shows when it will activate. These data structures are termed as branch counters, 'k'. The number of such counters appears to be $O(b^k)$. The worst case complexity of the third step is $O(j^{k+1} \cdot b \cdot k)$. Hence this step dominates the overall complexity. In practice, the worst case complexity depends heavily on the branching factor and the number of delay slots.

A model CFG is shown in Figure 3.4 for illustration. Further details about the CFG structure are explained as follows.

3.3.1. CPU and I/O Operations

Essentially, the CFG contains nodes of CPU related operations and peripheral (I/O) related operations. Here the number of I/O nodes and their positions are chosen arbitrarily leaving the rest as CPU-related nodes. In figure 3.4, the nodes that are marked as ‘I/O’ represent the section of CFG which involves the access to a particular I/O device.

Figure 3.4: Weighted CFG representing the Application with Peripheral Accesses

For illustrating the basic concept of the work, we take that the I/O nodes represent the I/O accesses attributed to one particular I/O device. In other words, those nodes represent the accesses of an I/O device which has multiple occurrences in an application.

Further, the nodes that are chosen as I/O access blocks can correspond to any I/O related services of an embedded application. Typical services include calls to the device drivers, DMA transfer, I/O mapped or Memory mapped peripheral- related operations.

Weighted Flow Graph

The nodes of the control flow graph are given weights which are directly related to the time taken for execution of each basic block. For simplicity, we assume that each basic block has fixed time duration for its execution and all edges have equal probability of transition. Hence, all outgoing edges from a particular node will have the same weights. They represent the least possible execution time of the block from which they are flowing out. This is shown in Figure 3.4.

In practice, we might get various execution times for the basic blocks which stem from a number of different reasons such as data dependencies, hardware effects such as pipelining and caching, etc. In such cases, the execution time estimate of a basic block for a particular target processor and the more frequent edges can be obtained by profiling runs and can be given as additional inputs to the algorithm.

The I/O access blocks denoted in the flow graphs are assumed to satisfy the timing needs of the peripheral device. The operations related to that particular I/O access block occurs within the duration of time assigned as its weight. In other words, the weight assigned to an I/O access block represents the time duration for which the peripheral needs to be in the operating state (ON).

3.3.2. Loops

The control flow graphs modeled in this work are *reducible* flow graphs. Before describing about reducibility of a graph, the basic concepts of dominators and natural loops need to be understood. A node d is said to be a dominator of node n , denoted “ $d \text{ dom } n$ ”, if every path from the initial node of the flow graph to n goes through d . Natural loop is a loop which has a single entry point, called the “header.” This entry point dominates all nodes in the loop. There must be at least one way to iterate the loop, i.e., at least one path back to the header.

A reducible flow graph is the flow graph in which there will be no jumps into the middle of loops from outside. The only entry to the loop is through its header. As such, the edges of such graphs can be partitioned into forward edges and backward edges. The forward edges form an acyclic graph in which every node can be reached from the initial node. The back edges consist only of edges whose heads dominate their tails. This is illustrated in Figure 3.5.

Figure 3.5: (a) CFG with back edges to the dominators (b) CFG in (a) “reduced” to a DAG with only forward edges

Loop Modeling

In a reducible flow graph, only natural loops need to be considered. Hence in our work, we model all our loops as natural loops. Further, natural loops with single entry and single exit are abstracted into a single block with its corresponding least possible execution time. Such natural loops that are abstracted as a single block are called as ‘loop block’ in this document.

Power management is the problem of predicting worst case idle time durations. Hence for any loop present, we need the least possible number of iterations. Obviously, for safety an iteration count of 1 for all loops can be assumed. But most embedded applications will have several loop bodies with non-trivial iteration counts. Hence, we assume minimum iteration bounds for all ‘loop blocks’ as additional information obtained from the CFG.

Figure 3.6: (a) Section of CFG having a Natural Loop with single exit (b) Abstracting the loop in with an equivalent ‘Loop Block’

The weight assigned to such a loop block would be the product of the least execution time of the whole loop block and the minimum iteration bound of that loop block. Denoting the minimum bound for loop iteration as MLI and the least execution time for a path within the loop as SP, the weight of the loop block LBW is given as:

Loop Block Weight = (Minimum Bound for Loop Iteration X Least Execution time).

This is illustrated in Figure 3.6.

The natural loops which have multiple exits are given the default iteration count of 1 as it will be complicated to deduce the order of inner loops execution and other issues

in determining the least number of iterations statically. Also, loops containing an I/O access point are not abstracted as single ‘loop block’ unless all individual blocks within the loop are I/O access blocks. This enables in utilizing the idle periods that may occur within that loop body which when abstracted might get missed out.

3.3.3. Procedure Calls

As most structural programming languages are written with various functions, a single thread of application may have several such functions. In this document, we use the words procedure calls/ sub-routines to denote functions. Most sub-routines will have non-trivial loops which might require high execution time durations. A procedure or sub-routine can be called through more than one path of the control flow graph. Hence every path has to account all the sub-routine execution times to estimate the net delay associated between any two nodes. It can be safely assumed that the execution time of a sub-routine does not depend on the path taken. In this work, we are interested in estimating idle times between successive I/O requests and take decisions for power management and this can happen at any arbitrary CPU-related block.

If a procedure or sub-routine block is selected for the decision-making, there arises an issue because the same procedure can be called by two or more paths which may not require the power management decision to be taken. Hence, there is a necessity to include a procedure or subroutine in every path which calls that routine. This problem can be tackled by inserting virtual blocks in the places of all subroutine calls. These virtual blocks are given weights based on the respective subroutine’s execution time.

Figure 3.7: (a) Section of CFG depicting Call to a Sub-routine (SR) (b) Replacing Sub-routine with an equivalent Virtual Block

The approach adapted here for handling the procedure calls is similar to that followed in the Worst-Case Execution Time (WCET) analysis of Hard-Real time systems. In a typical WCET analysis, a separate copy of the function is created at every distinct call site of the function in the program such that each call transfers control to its corresponding copy. Figure 3.7 represents the procedure-call modeling as explained above.

3.4. Summary

In this chapter, the significance of profile-based analysis for dynamic power management in embedded systems has been discussed. Employing application-aware strategy for the power management of peripherals utilizing control flow profile has been proposed. An outline of the proposed approach in this thesis was presented with an analogy. Further, the application model used in our work which is a weighted CFG was presented in detail with illustrative diagrams and equations.

In the next chapter, the proposed methodology is explained in detail with a two-pass algorithm. Enhancements to the basic algorithm to handle multiple devices and devices with multiple power-down modes will be discussed. Simulation analysis of the algorithm for each case will be presented. This will be followed by the evaluation of the proposed approach for overall efficiency. Also, techniques to manage the runtime overheads will be discussed as a means to contribute to the overall energy savings.

Chapter 4. Proposed Methodology for Power Management of Peripherals

The previous chapter presented an overview and analogy of our proposed approach and the application modeling. In this chapter, an algorithm has been devised for the proposed control-flow profile based technique for the dynamic power management of peripherals. It does the offline analysis of control flow graphs to take suitable power management decisions. The algorithm is then enhanced to deal with applications accessing multiple peripherals and also for peripherals with multiple power-down modes. The proposed approach is also evaluated for its efficiency and suitable techniques to manage overheads are also presented in the subsequent sections.

4.1. Overview

The dynamic power management scheme proposed in this thesis utilizes the control flow information that characterizes an embedded application with peripheral accesses. The previous chapter presented in detail the application model used in this work which is a weighted Control flow graph. Also, an outline of the proposed approach was presented. In this chapter, the proposed methodology is explained in detail with suitable illustrations and simulation analysis. An algorithm is devised to analyze static Control Flow Graphs for the power management of I/O devices. As such, this chapter presents the following:

- ❑ Detailed explanation of the proposed algorithm for profile-based power management of the peripherals. The two-pass algorithm is presented with illustrative diagrams and equations and the considerations upon which key decisions are taken. To illustrate the key concept of the proposed approach, only a single peripheral with multiple occurrences of its accesses is taken.
- ❑ Enhancements to the basic methodology to handle applications accessing multiple I/O devices and devices with multiple power-down modes. With the basic algorithm that dealt with single peripheral, suitable enhancements to

handle real scenarios such as a single thread of application with multiple I/O devices and devices with various power-down modes are presented.

- Simulation analysis and results for all the scenarios discussed. This is examined using a model Control flow graph representing the embedded application.
- Evaluation and efficiency analysis of the proposed approach. Discussion about the possible runtime overheads, techniques to handle those overheads and an estimation of the energy savings that can be obtained by employing this methodology are presented.

4.2. Algorithm to Analyze CFG for DPM

The application is modeled as described earlier with a CFG which has I/O access blocks at various arbitrary locations. To analyze the CFG, an offline algorithm has been devised to decide about the operating state of peripherals depending on their periods of activity and inactivity. This results in key dynamic power management decisions.

The algorithm consists of two key steps in two passes:

- 1) Identification of appropriate locations in the CFG to turn a device to ON state, known as “Switch-ON” points ensuring *performance*.
- 2) Identification of appropriate locations in the CFG to put a device into OFF state, known as “Switch-OFF” points ensuring *profitable* energy savings.

Further, a final elimination phase after these two passes help in eliminating any redundant switch-ON points.

4.2.1. Identification of Switching Points

In this work, the Switch-ON points are denoted as “*A*” points with “*A*” symbolizing “*Active*” state of operation and the Switch-OFF points are denoted as “*L*” points - with “*L*” symbolizing “*Low*” power idle state. Whenever the need arises to refer both of them, the term “*Switching points*” is used. When we refer as to “place” a switching point, what we mean here is to “identify” the CPU-related basic block in the CFG which favors the location of a control word command for switching the peripheral to ON or OFF state. Locating the switching point further within the basic block depends on the granularity of the block sizes considered and accordingly it can be split if it is

larger. For negligible block sizes compared to the device features, we can safely locate the switching point at the start of a block.

Identification of switching points for turning ON/OFF a device is done in two passes. The first one is the performance-centric identification of Switch-ON points and the other is the profitability-centric identification of Switch-OFF points. These two steps of this algorithm are elaborated in detail as follows:

4.2.2. Performance-centric Decision:

Identification of Switch-ON points (“A”) ensuring Performance: To start with, the peripheral under consideration is assumed to be in the OFF state. In the first pass of the algorithm, the CFG data structure is analyzed to get all necessary points for turning ON the peripheral. Each peripheral usage will have its own performance constraint as to within what duration of time, a peripheral has to respond for its access. Therefore, the time taken for a device to get turned ON, T_{ON} , determines the points to switch it ON. To turn it ON, a Switch-ON point (“A”) is “placed” along every path which leads to that I/O block at a location in the CFG which has “enough time” (i.e. T_{ON}) to Turn-ON the peripheral without performance penalty. This ensures that the device will be turned ON along all possible entries of an I/O access. Hence, the rule governing the placement of Switch-ON point becomes as follows:

$$\text{Required Time Duration between an “A” point and an I/O node} \geq T_{ON} \quad (4.1)$$

The above equation guarantees that the peripheral’s performance constraints are met satisfactorily as the device gets turned ON in time before its access. With this rule, the Switch-ON points are placed before every possible points of peripheral access, thereby ensuring that the device gets turned ON along all possible traces of its execution. As a result, there will be an “A” point before every I/O access unless the application itself doesn’t have enough time duration to cater T_{ON} needs. This first step of the algorithm is depicted in the following flowchart and the pseudo-code in Figures 4.1 & 4.2.

Figure 4.1: Flowchart Depicting the Placement of Switch ON “A” points (Step-1)

```
Procedure: place_A()
for all io ∈ IO
{temp x=io
iteration_1()
{ push all p∈P(x)
while (stack ≠ null)
{ p=pop
if p∉IO and x≠D(p)
if SD(p,io)≥TON
place "A" in block 'p'
else
x=p
call iteration_1()
}
}
}
```

Input: CFG, T_{ON} , T_{SE}
 $P(n)$: Immediate Predecessors of block 'n'
 $S(n)$: Immediate Successors of block 'n'
 $D(n)$: Dominators of block 'n'
 IO : Set of all I/O blocks
 $SD(m,n)$: Shortest distance from block 'm' to 'n'

Figure 4.2: First pass steps of the algorithm

Referring to Figure 4.2, the function “place_A” locates suitable positions in the CFG for the timely Switch-ON of the device. This makes the device to function without compromising on performance. Here only the basic flow is explained, as the details regarding exceptional cases arising in loops and procedures are discussed later in section 4.2.4.

Illustration

The Switch-ON points are placed before every possible entry to all peripheral accesses at a distance greater than or equal to T_{ON} , thereby ensuring that the device gets turned ON along every possible traces of the program execution. Figure 4.3 serves as an example to explain how the Switch ON points (“A”) are found out. Here the sizes of all blocks are taken to be equal. As shown, every path leading to the I/O block has an “A” point at a distance greater than or equal to T_{ON} .

Figure 4.3: Sample scenario to show the placement of “A” points

4.2.3. Profitability-centric Decision:

Identification of Profitable Switch-OFF points (“L”): In the second pass, now having known the locations of peripheral accesses and their Switch-ON points in the CFG, suitable points to turn OFF the peripheral for energy savings are identified. Among all possible paths from one I/O access to the successive I/O access, if the shortest path has “profitable time” ($> \text{Break-even time } T_{BE}$) to Turn-OFF the peripheral without power penalty, then a Switch-OFF point (“L”) is “placed” along every path from the current I/O block to the next I/O block. Hence, the rule governing the placement of Switch-OFF points is as follows:

$$\text{Required Time Duration between an “L” point and any I/Os} > T_{BE} \quad (4.2)$$

Since there is a possibility that once a device gets switched ON from a “A” point (placed in step-1), the program execution may take some other alternate routes of

Chapter 4. Proposed Methodology for Power Management of Peripherals

execution (which doesn't have a I/O within T_{BE}) leaving the "A" point unnecessary. Hence again, there arises a necessity to put an "L" point after identifying such alternate paths of execution.

Hence there are two types of Switch OFF points ("L"):

- 1) "L" point after I/O service
- 2) "L" point after an "A" point (if alternate paths of execution are followed)

Figure 4.4: Flowchart Depicting the Placement of "L" points

Algorithm- Second Pass

```

Procedure: place_L()
for all io ∈ IO
{
  temp y=io
  iteration_2()
  {
    push all s ∈ S(y)
    while (stack ≠ null)
    {
      s=pop
      if s ∈ IO and s ∉ D(y)
      if SD(s, IO) > TBE (check ∀ io ∈ IO)
        place "L" in block 's'
      else {
        y=s
        call iteration_2()
      }
    }
  }
}

```

Input: CFG, T_{ON} , T_{BE}
 $P(n)$: Immediate Predecessors of block 'n'
 $S(n)$: Immediate Successors of block 'n'
 $D(n)$: Dominators of block 'n'
 IO : Set of all I/O blocks
 $SD(m, n)$: Shortest distance from block 'm' to 'n'

Figure 4.5: Second Pass steps of the algorithm

Figure 4.4 & 4.5 summarizes the steps of the second pass of the algorithm. The function "place_L" identifies the positions for Switching-OFF of the device. The steps explained as in Figure 4.4 are performed on Switch-ON points also to check their

alternate routes of execution. Collectively, these decisions enables in getting profitable energy savings.

Illustration

As a device is turned off when it is idle, the energy savings obtained by turning it OFF should be justified. For that, the cost involved in turning it OFF and turning it ON again when it is required has to be compensated. Hence there will be a Switch OFF point, only if, there is a profitable time for energy savings before the next access of I/O takes place. The break-even time T_{BE} determines the minimum time duration between a Switch-OFF point and an I/O access. Figure 4.6 depicts a sample scenario for “L” points after an I/O service.

Figure 4.6: Sample scenario showing the placement of “L” points after I/O (a) Edge on the left not satisfying T_{BE} constraint- Hence no “L” point (b) Edge on the right satisfying T_{BE} constraint - Hence placing “L” Point

Since from step 1, the Switch ON points get placed before every peripheral access, there will be a Switch-ON point between a Switch OFF point and the next possible peripheral access along all paths of the program execution. This ensures that the device will be turned ON when it is required again without violating performance constraints. Also, since there is a possibility that once a device gets switched ON from

an “A” point (placed in step-1), the program execution may take some other alternate routes of execution leaving that “A” point unnecessary.

Hence again, there arises a necessity to put an “L” point after identifying such alternate paths of execution. Figure 4.7 depicts this scenario. This is also one of the reasons for considering the switch-OFF points in the second step of the algorithm.

Here, the additional constraint in placing an “L” point after an I/O is that it should ensure that the device has already turned ON as the result of the previous “A” point. Hence it should account for the device turn-ON time T_{ON} . This constraint can be ensured by checking the distance between “A” point and any I/O in the alternate path taken is greater than $(T_{BE} + T_{ON})$.

Figure 4.7: Sample scenario showing the placement of “L” points after “A” points: (a) “A” point on the left leads to I/O- Hence no “L” point (b) “A” point on the right may not always lead to I/O -Hence placing “L” point

4.2.4. Elimination Phase for Redundant Switch-ON points:

As explained in the previous sections, from the two passes of the algorithm, the switch-ON and switch-OFF points are identified. In the first pass of the algorithm, the Switch-ON points are placed along every possible entries to an I/O access node based on the Turn-ON time of the device. However, this gives an exhaustive list of Switch-

On points. This is to consider the fact that a device may get turned OFF anywhere in the control flow and might enter along any possible entry to the I/O access.

However, in the second pass, we get to know all possible locations of Switch-OFF points in the CFG. With this information, a suitable algorithm is devised to determine whether a Switch-ON point is redundant or not. Once the redundant points are identified, the switch-ON status is *eliminated* from them and they are not switch-ON points of the device anymore. This is termed as the “Elimination Phase” after the two passes of the algorithm, as this gets rid of the identified redundant Switch-ON points.

The cases for this elimination phase are presented as follows:

Case-1:

A Switch-ON point ‘X’ is redundant if both the following conditions are satisfied.

- i. The Switch-ON point ‘X’ is dominated by another Switch-ON point ‘Y’ before getting dominated by a Switch-OFF point in the dominator tree.
- ii. There are no paths between any Switch-OFF points dominated by ‘Y’ to ‘X’.

This is illustrated in Figure 4.8.

Figure 4.8: Cases of Elimination of “A” when dominated by another “A”

Case-2:

Similarly, a Switch-ON point 'Z' is redundant if both the following conditions are satisfied.

- A Switch-ON point 'Z' is dominated by an I/O access node 'IO' before getting dominated by a Switch-OFF point in the dominator tree.
- There are no paths between any Switch-OFF points dominated by 'IO' to 'Z'.

This is illustrated in Figure 4.9 while Figure 4.10 gives the basic pseudo-code of the elimination phase.

Figure 4.9: Cases of Elimination of "A" when dominated by an "I/O"

```

for all 'X' ∈ {IOA}
{
  if IDom(X) ∈ {IOA}
    for all 'i' ∈ {L_X}
    {
      if SP('i', 'X') is not valid
      {mark 'X' as redundant.
       Remove 'A' from 'X'.}
    }
}

```

{A} : Set of all Switch-ON points {IOA}: Union of all I/O nodes and Switch-ON points {L_X}: Set of all Switch-OFF points dominated by 'X' Idom(X): Immediate Dominator of 'X' SP(m,n): Shortest path between nodes m,n.

Figure 4.10: Basic steps in the elimination phase

4.2.5. Handling Loops & Procedure Calls

This section explains about dealing with loops and procedure calls of an application code.

Loops

While calculating the shortest path from any switching point to an I/O access, the loops with multiple exits are safely given a minimum loop bound of 1. In practice, if the minimum loop bounds for such loops are known, these can be given as additional inputs and accordingly the shortest path algorithm can be modified to account this additional data.

Figure 4.11: (a) “A” point in a ‘Loop Block’ characterized by Monitoring Count (b) Switching Points utilizing idle periods within a loop

The loops which are abstracted as loop blocks are given a non-trivial iteration value. There might occur scenarios where such a loop block be selected as a Switch-ON point. In such case, an approximate estimate of the iteration value that would be sufficient enough to cater the Turn-ON time requirements is calculated. Such an estimate of the number of iterations can be termed as ‘monitoring count’. Hence for

Switch-ON points that fall within a loop can be characterized by both their locations and the corresponding ‘monitoring count’.

In certain cases, there may be an I/O access which falls within a loop but has profitable idle time duration for energy savings. Hence a Switch-OFF point can be placed after its service. But the looping edge may take inside the loop without a Switch-ON point on its way. This is handled by placing a Switch-ON point within the loop before the back edge takes to the I/O access. This is depicted in Figure 4.11.

Procedure Calls

If a procedure or subroutine is selected as the switching point, there arises an issue because the same procedure can be called by two or more paths which may not require the Switch-ON/OFF decision to be taken. Hence, as mentioned earlier, a virtual block of corresponding weight acts as the procedure sub-routine. Once a virtual block gets selected as the switching point, then the predecessor or the successor of that virtual block is chosen instead. Depending on the case and application scenarios, such decisions could be taken with an associated penalty in either power or performance.

Figure 4.12: (a) Virtual Block getting selected as Switching Point (b) Shifting the Switching Points to nearer appropriate locations.

Figure 4.12 depicts the scenario of dealing with the case when a procedure call which has been represented as virtual block is selected as the switching point.

4.3. Simulation Analysis of the Basic Algorithm:

This section presents the simulation analysis of the basic methodology of the proposed approach using a model control flow graph. The basic algorithm is presented in view of a single thread of application accessing a single peripheral responding to hard Turn-ON/OFF i.e. either a device can be in the ON state or OFF state with no other intermediate modes.

Generation of Random CFGs:

To generate random CFGs representing the application model as described in the previous chapter, a software program was written using the random number generation function and the “Biograph” toolbox in MATLAB.

- ❑ The level of randomness in the generation of CFGs was controlled using the following parameters: (1) Maximum depth level of a CFG which is chosen from 10 to 20. (2) Minimum and maximum values for the number of predecessors/successors of a node which is chosen from 2 to 5. This controls the breadth of each level. (3) The depth level of an interconnection between the nodes. The ranges are chosen so as to give an aesthetic visibility for analysis, though the algorithm is independent of such ranges. The “randint” function in MATLAB helped in generating the random numbers in these predefined ranges resulting in different number of nodes, interconnections and natural loops.
- ❑ To ensure only natural loops, all back edges are chosen to point back only to their dominators. In other words, the nodes at a given depth should branch back always to the higher level nodes which dominate them. Further nested loops are not considered.
- ❑ Here the size (measure of execution time duration) of all nodes in the CFG is assumed to be uniform throughout the graph and can be varied as per the inputs. Though the proposed algorithm doesn’t depend on this assumption, it serves for simplicity.

CFG used for Illustration:

The CFG used here consists of 67 nodes with random interconnections as explained above. Though the algorithm holds good for any number of nodes in the CFG, here the number of nodes are restricted for clarity and ease of understanding while illustration.

Here the time taken for execution of each block is taken to be of 1 millisecond duration for all illustrations. Figure 4.13 shows this model CFG.

Regarding the pictorial representation

- ❑ All normal CPU blocks are represented as regular rectangular boxes.
- ❑ The I/O access blocks are shown as shaded rectangular boxes.
- ❑ The resulting Switch ON points and Switch OFF points obtained by running the simulation are represented in elliptical boxes marked as “A” and “L” respectively.

Figure 4.13: Model CFG used for Basic Algorithm Simulation

Simulation Analysis

An application accessing a single instance of an I/O device at various points of its execution is considered. In the CFG taken, first the I/O access points are chosen randomly as (Blocks-4, 5, 6, 23, 34, 36, 37, 53, 60, 61) and shaded to mark their difference as shown in Figure 4.13.

Citing the equations governing this algorithm again here for convenience:

Required Time Duration between an “A” point and I/O $\geq T_{ON}$

Required Time Duration between an “L” point and any I/Os $> T_{BE}$

For the simulation analysis, the values for important parameters are assigned as: Block Size (CFG) = 1ms; T_{ON} =1ms; T_{BE} =3ms. The steps of the algorithm in the simulation are summarized here. The graphs with the results obtained in each steps are shown in the appendix section. Only the final output resulting from all steps have been presented here in Figure 4.14.

1) Performance-centric Switch-ONs: The points to turn ON i.e. “A” points are needed to be identified first. The equation (4.1) governing this rule now gets translated (for this particular example) as follows: Along every possible entry of an I/O access point, there will be an “A” point at a distance of a single block size (1 ms). Hence the “A”s obtained from running this algorithm by simulating in Matlab are as follows: Blocks numbered (2,3,17,18,29,32,47,48,55,56,57,58) as shown in Figure A.1 in appendix.

2) Profitability-centric Switch-OFFs: As the second step, the points to turn OFF the device for profitable energy savings are identified which is governed by equation (4.2). Their interpretation in this scenario will be: “L” points are placed once the device completes its operation and infers that there are no any immediate requirements of its functions for a total time duration of the execution of more than three blocks along any of its future paths of execution. Also, because of the previous step (i.e. performance-centric switch-ONs), there might be instances where a device was switched ON anticipating its requirement but would have followed some other paths of execution. Hence, there is a need to put an “A” in such scenarios too, once the alternate route taken is identified. Overall, the “L” points identified are Blocks (7, 8, 9, 10, 11, 26, 39, 40, 41, 59, 62, 63, 64, 67). The CFG with the switch-OFF points (“L”) identified can be seen in Figure A.2 of appendix.

3) Elimination Phase for redundant switch-on points: As explained earlier, the first step of performance-centric switch-ON identification, gives an exhaustive list of “A” points to ensure performance and proper functioning of the device along all paths. But with the knowledge of the locations of “L” points from step-2, redundant “A” points are eliminated with the simple algorithm explained in section 4.2.4. Hence, the “A” points @ Blocks (3, 17, 18, 58) are eliminated.

The final list of switching points obtained from the simulation is shown in Figure 4.14.

Complexity Analysis of the Basic Algorithm:

When studying the complexity analysis of the proposed algorithm to analyze the CFG, we infer that the total number of nodes in a CFG, say, ‘n’ and number of I/O nodes, say, ‘m’ are the determining parameters.

The algorithm starts the analysis wrt to an I/O node and it does the identification of relevant switching points by searching the neighborhood predecessor and successor nodes. As such, this process is repeated for every I/O node. In the worst case, every node among the ‘m’ I/O nodes has to undergo a search of all ‘n’ nodes. Since this is a two-pass algorithm, both the passes undergo similar steps.

Hence, the complexity becomes $2*m*n$. In Big Oh notation, the complexity can be expressed as $O(m*n)$.

Chapter 4. Proposed Methodology for Power Management of Peripherals

Figure 4.14: Basic Algorithm Simulation –Final List of switching points

4.4. Applications Accessing Multiple I/O Devices:

The previous sections examined the basic algorithm in view of a single thread of application accessing a single peripheral responding to hard Turn-ON/OFF. In this section, enhancement to the same algorithm to applications accessing multiple devices is presented. Simulation results for a model CFG is presented.

A real application might involve access to more than one peripheral. This may occur even within a single thread of execution. The same algorithm can be used for applications accessing multiple peripherals if they are independent of each other.

4.4.1. Multiple Sets of Switching Points

For the case of multiple I/O devices, its CFG is first identified with all types of I/O accesses and can be categorized. The algorithm then will give as many sets of switching points as the number of peripherals. In some cases, a switching point may happen to place in an I/O access block of some other devices. To circumvent this issue, the switching point is selected as either the predecessor or successor with a certain penalty in performance or power. Hence the set of rules governing the placement of switching points will vary accordingly which is elaborated as follows.

Enhanced Algorithm for Multiple I/O Devices

The same algorithm is enhanced accordingly to deal with applications accessing multiple I/O devices. The important considerations are presented as follows:

1. Each peripheral's operation is assumed to be independent that of the other.
2. In addition to estimating the execution time of CPU-related blocks in the CFG, a worst-case estimation of all I/O related nodes of all peripherals needs to be done. This is to make use of the time spent in a peripheral operation which serves as the idle time duration for another peripheral. If the time duration of a peripheral operation can't be determined, the default value assumed is zero.
3. For a Switching point to get placed, it needs to satisfy the usual governing equations mentioned earlier and additional constraint that the Switching point should not be an I/O access point of any other devices.

The steps followed in the algorithm are given in Figure 4.15.

The modified sets of equations are as follows:

$$\begin{aligned} \text{Required Time Duration between an "A" point and corresponding type I/O} &\geq T_{ON} \\ \text{AND ("A" } \notin \{IO\}) & \end{aligned} \quad (4.3)$$

$$\begin{aligned} \text{Required Time Duration between an "L" point and} \\ \text{next occurrence of same type "I/O"s} &> T_{BE} \\ \text{AND ("L" } \notin \{IO\}) & \end{aligned} \quad (4.4)$$

Algorithm- Multiple I/O Case

```

Iterate (1: Number of devices)
for all io_d ∈ IO_d
{temp x=io
iteration_1()
{ push all p ∈ P(x)
while (stack ≠ null)
{ p=pop
if p ∉ IO_U and x ∉ D(p)
{ if SD(p,io) ≥ T_ON
place "A" in block 'p'
else
x=p
call iteration_1()
}
}
}
}

```

Input: CFG, T_{ON} , T_{BE}
 $P(n)$: Immediate Predecessors of block 'n'
 $S(n)$: Immediate Successors of block 'n'
 $D(n)$: Dominators of block 'n'
 IO_n : Set of all I/O blocks of device 'd'
 IO_U : Union of all I/O blocks of multiple devices
 $SD(m,n)$: Shortest distance from block 'm' to 'n'

Figure 4.15: Enhanced Algorithm for Multiple I/O case

4.4.2. Simulation Analysis

To analyze the application with multiple I/O devices, its CFG is first identified with all types of I/O accesses and then categorized. For illustration purposes, the type number is marked on the respective I/O access blocks of the CFG. All other assumptions regarding the CFG holds good for this case too. To illustrate here the sample CFG is chosen to have two different I/O devices with their corresponding access nodes spread randomly.

Starting with an example as was done for single I/O case, the device features of all peripherals can be taken as the same for simplicity. The values for important parameters are assigned as: Block Size (CFG) = 1ms; T_{ON} =1ms; T_{BE} =3ms. Following the two-step methodology of this algorithm for an application with multiple I/O devices:

Similar to the basic algorithm and with few additional constraints, the steps of the algorithm is explained as follows:

- ❑ **Performance-centric Switch-ONs:** The points of respective turn ONs (“A”s) are identified first for each I/O device ensuring performance using the TON parameter. Also the identified block should be ensured as a non-I/O block of any type. Hence there will be as many sets of “A”s as the number of I/O devices.
- ❑ **Profitability-centric Switch-OFFs :** Incorporating the above additional constraint in the second step too, the points to turn OFF every I/O device once they are idle and once alternate routes of execution following an “A” point are identified. Here, the break-even time of individual devices is the deciding parameter.
- ❑ **Elimination Phase:** The final elimination phase strategy is applied to eliminate redundant switch-ON points for all the devices.

Representing Multiple Sets of Switching points:

The simulation is run with CFG identified with different I/O access nodes corresponding to the two devices. As a result, we get as many sets of Switching points as the number of I/O devices in the application considered. As a switching point denotes a location in the control flow profile, the same location can be the switching point of either type (i.e. ON/OFF) for more than one device. Hence whenever an application with multiple I/O devices is dealt with, a switching point needs to be characterized by both the switching states and the device number.

Representation of switch-ON points of device-n: “nA”

(E.g. “1A”, “2A”, etc.)

Representation of switch-OFF points of device-n: “nL”

(E.g. “1L”, “2L”, etc.)

The simulation analysis results are shown in two stages in Figures A.4 & A.5 in the appendix:

1. Identification of all switching points of device-1
2. Identification of all switching points of device-2 which gets superimposed on results of device-1.

4.5. Peripherals with Multiple Power-Down States:

A device may support multiple power-down states each having its own power consumption level and break-even time. States are the various modes of power down levels that span the tradeoff between performance and power. Each transition between different modes of operation has a cost. In many cases, the cost is in terms of delay or performance loss and sometimes there may be a transition power cost. Hence, the devised algorithm is extended for peripherals with multiple power-down states. For a device with multiple power-down modes, every mode will have its own break-even time constraints. With this notion, the enhancements to the algorithm are presented in this section.

When a device supports multiple power saving modes, correspondingly there exists as many sets of break-even and Turn ON times. For example, some hard disks have a standby state and a sleeping state. These disks consume less power in their sleeping states compared to the standby states. However, a sleeping disk requires a hardware reset to wake up; a standby disk does not need resetting. Figure 4.16 illustrates the differing timing requirements for various state transitions of a virtual device supporting multiple power-down modes.

Figure 4.16: Illustrating variable parameters for multiple power-down states

4.5.1. Enhanced Algorithm for Multiple Power-Down States

For a peripheral device supporting multiple power-down modes, the different sets of parameters appropriate for every mode need to be considered. The basic methodology can be dealt in a modified manner as follows:

1. The algorithm needs to be iterated for every pass as many times as the number of power-down modes.
2. Using the basic algorithm, the performance-centric Switch-On points are found as usual, using the corresponding value of Turn-ON time for that mode in the first pass.
3. Similarly, with the different break-even time parameters applicable to each state, the Switch-Off points for every mode are found in the corresponding iteration.
4. Finally, a modified elimination phase is applied with the available sets of data. This involves determining the necessary sets of Switch-ON points for every mode.

The detailed methodology is explained as follows. The enhanced algorithm for considering multiple power-down states consists of the following three steps:

- 1) Identifying different sets of Switch-ON points
- 2) Identifying different sets of Switch-OFF points
- 3) Eliminating redundant Switch-ON points of various modes

As the more general case, here, we consider an application accessing a single device which has multiple power-down modes.

1) Identifying different sets of Switch-ON points:

Given a set of power-down modes for a peripheral, there will be as many sets of Turn-ON time and Break-even time parameters. This leads to different sets of Switch-ON points and Switch-OFF points in our proposed methodology.

The performance-centric first pass of the algorithm is applied with a modification which is explained as follows. Let $T_{ON1}, T_{ON2}, \dots, T_{ONn}$ etc. ($T_{ON1} < T_{ON2} < \dots < T_{ONn}$) denote the turn-ON times to put the device into ON state from the various corresponding power-down modes. The algorithm is iterated as many times as the number of power-down modes with corresponding values of Turn-On times ($T_{ON1}, T_{ON2}, \dots, T_{ONn}$). This enables in getting the various sets of Switch-ON points for every mode.

Let Sw_ON_1, Sw_ON_2, Sw_ON_3, etc. denote the different sets of Switch-ON points which put the device into the ON state for the corresponding power-down modes. This set gives an exhaustive list of all possible Switch-On points before all possible entries of an I/O access. The problem of avoiding the redundant Switch-On points and retaining only those necessary is dealt in the elimination phase discussed in the subsequent section.

2) Identifying different sets of Switch-OFF points:

As mentioned earlier, the Switch-OFF points are of two types: the first type is placed after the I/O service and the other type is placed after an unused Switch-On point obtained in the first pass. Hence before proceeding with the second pass of identifying the Switch-OFF points, the Switch-ON points to be considered is the union of Switch-On points of all various modes.

Let Sw_OFF_1, Sw_OFF_2, Sw_OFF_3, etc. denote the different sets of Switch-OFF points which put the device into the various corresponding power-down modes. Let T_{BE1} , T_{BE2} , T_{BE3} , etc. ($T_{BE1} < T_{BE2} < T_{BE3}$) denote the corresponding break-even time parameters for those various power-down modes.

With the following notations:

S(n): Immediate Successors of block 'n'

IO : Set of all I/O access blocks of a peripheral in the CFG.

SD (m , n): Shortest distance from block 'm' to 'n'

Sw_ON_U: Set of union of all Switch-ON points of all modes =

$\{Sw_ON_1\} \cup \{Sw_ON_2\} \cup \dots \cup \{Sw_ON_n-1\} \cup \{Sw_ON_n\}$

("U" denotes set union operation).

A Switch-OFF point belonging to a particular mode is chosen only if it satisfies the following criteria,

$SD(\text{Successors of 'io'} \in IO, \text{ all other I/O nodes}) > T_{BE_n} \ \& \ < T_{BE_{(n+1)}}$

$SD(\text{Successors of 'u'} \in Sw_ON_U, \text{ all other I/O nodes}) > T_{BE_n} \ \& \ < T_{BE_{(n+1)}}$

In other words, while traversing from either the successors of an I/O node or the Switch-On points, the current node under consideration is chosen as a Switch-OFF

point for mode 'n' only if the shortest distance to the nearest I/O node is greater than the break-even time for mode 'n' but less than the break-even time for mode 'n+1'.

3) Eliminating redundant Switch-ON points of various modes:

In the first pass, with every iteration of the algorithm, there will be different sets of Switch-ON points corresponding to each mode. In the second pass, the different sets of Switch-OFF points corresponding to each mode are obtained. Now with this data, a modified elimination phase resembling the one dealt in section 4.2.4 is employed with few special constraints to retain only the necessary Switch-ON points for every mode. This is explained as follows:

In the first pass, the Switch-ON points corresponding to all power-down modes are placed along every possible entries to an I/O access node based on the Turn-ON time specific for that mode. This gives an exhaustive list of Switch-On points for all modes. This is to consider the fact that a device may be in any one of the power-down mode anywhere in the control flow and might enter along any possible entry to the I/O access. Hence the redundant Switch-ON points belonging to all modes are eliminated using the following strategies.

With the previous notations:

Sw_ON_U: Set of union of all Switch-On points of all modes = {Sw_ON_1, Sw_ON_2,...}

Sw_ON_n: denotes the set of Switch-On points corresponding to mode 'n'.

Sw_OFF_n: denotes the set of Switch-OFF points corresponding to mode 'n'

Among the set of Switch-ON points of type 'n' i.e. Sw_ON_n, a point is retained only if it has at least one path *directly* from a Switch-OFF point of the same type 'n' i.e. Sw_OFF_n. Here the term *direct path* is used to refer a path from a Switch-OFF point to a Switch-ON point both belonging to the same mode without including a Switch-OFF point of any other modes having higher break-even times.

The following methodology is applied to determine the conditions under which a Switch-ON point belonging to a particular mode can be retained or eliminated.

For mode- n which has the highest Break-even time:**Case-1:**

A Switch-ON point of mode-1 ' X_n ' is redundant if both the following conditions are satisfied.

- i. The Switch-ON point ' X_n ' is dominated by another Switch-ON point ' Y_n ' before getting dominated by a Switch-OFF point ' $off_n \in \{Sw_OFF_n\}$ ' in the dominator tree.
- ii. There are no paths between any Switch-OFF points ' $off_1 \in \{Sw_OFF_n\}$ ' dominated by ' Y_n ' to ' X_n '.

Case-2:

Similarly, a Switch-ON point ' Z_n ' is redundant if both the following conditions are satisfied.

- i. A Switch-ON point ' Z_n ' is dominated by an I/O access node 'IO' before getting dominated by a Switch-OFF point in the dominator tree.
- ii. There are no paths between a Switch-OFF point ' $off_n \in \{Sw_OFF_n\}$ ' dominated by 'IO' to ' Z_n '.

For all other modes:

For all other modes, i.e. mode- $(n-1)$,.....mode-1 which have break-even times in decreasing order, the following methodology is applied.

Case-1:

A Switch-ON point of mode- p ' X_p ' is redundant if both the following conditions are satisfied.

- i. The Switch-ON point ' X_p ' is dominated by any other Switch-ON point ' $Y_u \in \{Sw_ON_p \cup Sw_ON_{(p+1)} \cup \dots Sw_ON_n\}$ ' before getting dominated by any Switch-OFF point ' $off_u \in \{Sw_OFF_p \cup Sw_OFF_{(p+1)} \cup \dots Sw_OFF_n\}$ ' in the dominator tree.
- ii. There are no paths between any Switch-OFF points ' $off_p \in \{Sw_OFF_p\}$ ' (of the same mode type) dominated by ' Y_p ' to ' X_p '.

Case-2:

Similarly, a Switch-ON point 'Z_p' is redundant if both the following conditions are satisfied.

- i. A Switch-ON point 'Z_p' is dominated by an I/O access node 'IO' before getting dominated by any Switch-OFF point 'off_u' $\in \{Sw_OFF_p \cup Sw_OFF_p+1 \cup \dots Sw_OFF_n\}$ in the dominator tree.
- ii. There are no paths between any Switch-OFF points 'off_p' $\in \{Sw_OFF_p\}$ (of the same mode type) dominated by 'IO' to 'Z_p'.

Thus, the above strategies are built-on as enhancements to the general algorithm to handle devices with multiple power-down states.

A constraint in handling a device with multiple power-down states is that the break-even time of the least power-down mode needs to be greater than the turn-ON time of the highest mode ($T_{BE1} > T_{ONn}$). Otherwise, there is a possibility for overlapping of switch-OFF points of lower power-down modes and switch-ON points corresponding to any other higher power-down modes. Hence this may result in either performance or power penalty depending on the switching point we choose to have. This limitation in handling device with multiple power-down states can be handled with certain knowledge of the frequency of paths taken by program execution through some profiling techniques.

4.5.2. Simulation Analysis

With this enhanced algorithm, the model control flow graph chosen earlier is again injected with random I/O access nodes. As a simpler case to represent, a device supporting two power-down states is chosen. The parameters for these states are taken as: Block Size (CFG) = 1ms; $T_{ON1}=1ms$, $T_{BE1}=2ms$; $T_{ON2}=2ms$, $T_{BE2}=4ms$;

The simulation analysis is shown in two stages in Figures A.6 & A.7 at the appendix.

1. The results for the highest power-down mode (i.e. mode-2) applying the usual strategy.

2. Identification of switching points of the lower mode (i.e mode-1) after applying the enhanced algorithm. This gets superimposed on the previous results of higher order modes.

Representing multiple types of Switching points:

The simulation is run with CFG identified with different I/O access nodes of a single device that supports two power down modes. As a result, we get as two types of Switching points corresponding to the number of power down modes. To distinguish the two type, a switching point needs to be characterized by both the switching states and corresponding power-down mode to which it belongs.

Representation of switch-ON points of mode-n: “A_n”

(E.g. “A_1”, “A_2”, etc.)

Representation of switch-OFF points of mode-n: “L_2”

(E.g. “L_1”, “L_2”, etc.)

Having discussed with the basic methodology of the algorithm and its suitable enhancements, in the following sections, the evaluation of the proposed methodology is presented with discussion about possible overheads and suitable techniques to manage them.

4.6. Evaluation of the Proposed Approach

In this section, the proposed approach is evaluated for the energy savings that can be obtained by the estimation of the potential idle time durations. Discussion about possible runtime overheads and techniques to mitigate their effects to gain overall energy savings are also presented.

4.6.1. Estimation of Potential Idle Time Durations:

In this section, a discussion about the estimation of worst-case potential idle time duration for energy savings is presented.

As observed, the worst-case potential idle time duration for energy savings depends upon the shortest time duration taken by a Switch OFF point to meet the nearest Switch ON point. Hence, the energy savings is much dependent upon the density of

I/O access blocks. Less closer they are, more opportunities for energy savings. Hence an application with relatively larger distances among consecutive I/O access blocks along several paths of execution can gain much from this approach.

The time duration between a Switch OFF point and the nearest successive I/O access block along a path is a measure of the idle time between successive I/O blocks. Denoting that quantity as T_{SI} , the time duration that proves fruitful for energy savings (T_{ES}), will be the difference between T_{SI} and the break-even time (T_{BE}).

$$T_{ES} = (T_{SI} - T_{BE}) \quad (4.5)$$

The cumulative sum of T_{ES} along a dynamic trace of execution gives a quantitative measure of the energy savings. Considering percentage energy savings for an average of T_{ES} ,

$$T_{ES} / T_{SI} = (1 - T_{BE} / T_{SI}) \quad (4.6)$$

Defining the ratio of T_{BE} to T_{SI} as the “density index” d ,

$$T_{ES} / T_{SI} = (1 - d), \quad 1 > d > 0. \quad (4.7)$$

Figure 4.17: Percentage Energy Savings Vs Sparseness of I/O accesses

The quantity “d” gives us a measure of the “density” of I/O access blocks in a CFG. Lesser the value of “d”, sparser are the I/O access blocks favoring the applicability of this approach. Figure 4.17 represents the equation 4.7 graphically.

For applications accessing multiple peripherals, the energy savings achieved for the individual devices contribute to the cumulative energy savings of the system. For devices with multiple power-down states, an application having I/O access nodes placed more denser also can find opportunities for energy savings. Each path has the possibility to support a switching point of any power-down state ensuring a minimum energy savings.

Average Energy Savings:

To get rapid estimation of average energy savings using this approach, suitable offline profiling techniques can be used. This identifies the criticality/frequency of various paths. Hence, a few key paths can be chosen and the energy savings that can be obtained along each path can be found and then their average can be calculated. Also, when considering such key critical paths, support for multiple power modes will contribute effectively to average energy savings.

4.6.2. Managing Overheads for Efficiency:

The switching points found offline aids in the timely shut-down and powering the device for power management during runtime. Hence, there will be some additional work done in executing the control words for these switching points during program execution. Hence there is an associated cost function for these overheads incurred by the processor/operating system kernel of the target platform.

By assigning a cost function to these overheads, these can be accounted during the offline analysis. This can be done by varying the input parameters of the algorithm. For instance, the energy overhead costs can be expressed in terms of increased break-even time in the algorithm. Further, the latency can affect the performance demanding the TON parameter to be higher. This improves the efficiency of the overall approach. This also helps in knowing the feasibility and applicable scenarios of this proposed method. Here, as we discuss about the algorithm part, we give an outline for

accounting these overheads. In subsequent chapters, when we discuss about implementation strategies, we elaborate on this further.

Yet another approach to minimize overheads is to choose an optimum set of “Switching points” which have relatively more significant energy savings to justify the overheads. By choosing only an optimum set of significant switching points the overheads can be reduced further. This is further elaborated in the following section.

4.6.3. Selection of Significant Switching Points:

As observed, a scalable option in the number of switching points identified can prove to be an effective solution to minimize the overheads. Here, we devise a strategy to select switching points based on their *Significance*.

With this proposed method, the worst-case potential idle time duration for energy savings depends upon the shortest time duration taken by a Switch OFF point to meet the nearest Switch ON point. But a Switch-OFF point can lead to an I/O access through paths of various time durations. Hence, there is a need for a strategy to infer the importance of a given Switch-OFF point for power management. Hence we define the term “*Significance*” of a Switch-OFF point.

Significance of a Switch-OFF point:

1. The significance of a Switch-OFF point is quantified as the weighted average energy savings that can be obtained by the Switch-OFF point before the next I/O access.
2. The weighted average is the energy savings (i.e. idle time durations) of all individual paths from the Switch-OFF point to the next I/O access where the weights to each individual path depend on the frequency/criticality of the path. For this work, we are assuming equal weights for all paths. Hence this is simply the average quantity.

In calculating the idle time durations of all possible paths from the Switch-OFF point to the next I/O access, there is a possibility for higher complexity in computations. This is because the number of all possible simple paths between two nodes in a graph increases exponentially as the number of nodes in the graph increases. Hence as the

number of nodes in a graph becomes increasingly large, the computation complexity becomes much higher [50]. Therefore, a suitable engineering approach is to circumvent this issue is to approximate the average energy savings by taking the average of a few random paths, say 10.

As an illustrative example, a model control flow graph of 60 nodes is chosen as shown in Figure 4.18. The block size of each node is assigned as 10 ms. The simulation of the algorithm is run to find all the Switching points. Now, the Switch-OFF points are categorized based on the significance as defined earlier. They are tabulated in the decreasing order of their significance as shown in Table 4.1. Here, we consider only those switch-OFF points which lead to the neighboring I/O accesses. Since it's a graph of nominal size with 60 nodes, we have calculated the average idle time durations accounting all possible paths instead of approximating. The second column in the table refers to the neighboring switch-on points from the Switch-OFF point of first column.

Block Size=10 ms; T_{ON} =10 ms; T_{BE} =20 ms

(a) Listing all switching points

(b) Selecting only significant switch-off points and eliminating redundant switch-On points

Figure 4.18: Selection of significant switching points

Switch-OFF points leading to an I/O service (Block No)	Neighboring Switch- ON points (Block No)	Significance (Average Energy Savings: Idle time in ms)
8	39,40	60
9	39,40	50
6	26,27	40
7	26,27	40
33	44	20
34	44	20
35	44	20
43	53,54	20
45	53,54	20
46	53,54	10

Table 4.1: Ordering Switch-OFF points based on their significance

Eliminating less significant Switching points:

Having estimated the significance of a Switch-OFF point, a “threshold value” can be assigned for the *Significance* which can be fixed based on the application under examination. Now for the example above, say all Switch-OFF points with a Significance value which is greater than or equal to 40 only need to be considered. Then this leads to the elimination of the Switch-OFF points numbered as 33, 34, 35, 43, 45, 46 (as grayed out in the table 4.2).

Switch-OFF points leading to an I/O service (Block No)	Neighboring Switch- ON points (Block No)	Significance (Average Energy Savings: Idle time in ms)
8	39,40	60
9	39,40	50
6	26,27	40
7	26,27	40
33	44	20
34	44	20
35	44	20
43	53,54	20
45	53,54	20
46	53,54	10

Table 4.2: Selection of Switch-OFF points based on their significance

Now with a few Switch-OFF points getting eliminated, we again apply the **Elimination phase** for eliminating redundant Switch-On points. This leads to the elimination of those Switch-On points which were dependent on the now-eliminated Switch-OFF points. Applying the algorithm for the elimination phase, we get those “A” points at blocks numbered as (44, 53, 54) as the redundant Switch-ON points which can be eliminated.

The following figure 4.19 summarizes the general flow of our methodology incorporating the offline analysis and evaluation for its performance and efficiency.

Figure 4.19: General flow of the proposed methodology

4.7. Summary

In this chapter, the proposed profile-based approach for dynamic power management of the peripherals in embedded system applications was explained in detail. This involves manipulating the control flow information of an embedded application. The algorithm finds appropriate locations in the CFG for placing the switching points for power management of the peripherals. The performance-centric decisions enable in locating all the Switch-ON points. They are necessary for the timely operation of the device ensuring performance. Once the device has completed its service, the algorithm finds for profitable idle time durations that can ensure energy savings. This determines the location of the Switch-OFF points in the program execution. Techniques to handle typical applications with loops and procedures were discussed.

Subsequent sections dealt with the enhancements to the algorithm with simulation results. They include the incorporation of the methodology for general embedded applications dealing with more than one peripheral device and device with multiple power-down states. The algorithm handles both applications accessing single and multiple peripheral devices. Peripherals with multiple power-down modes can also be dealt with an enhanced version of the algorithm to have significant energy savings.

Further, the proposed methodology is evaluated for its efficiency. Typical estimation of energy savings with this proposed method was presented. It is evident that significant energy savings can be obtained from this approach depending upon the density of successive I/O access blocks in the application. Techniques to manage the overheads and to improve the efficiency were presented. Strategy to choose significant switching points was presented. Figure 4.19 summarizes our proposed methodology graphically.

The following chapter delves into the discussion of various possible implementations of the proposed approach. As such, the next chapter proposes a power-efficient hardware-assisted approach to monitor and control the peripherals for dynamic power management.

Chapter 5. Hardware-Assisted Monitor & Control for DPM

In this chapter, first the various possible implementation schemes of the proposed methodology of previous chapter and the associated issues that need to be considered are discussed. This includes the representation of the data obtained as a result of the offline analysis and analysis of efficient strategies to implement with minimum overheads. We then investigate the design of a custom hardware for DPM that incorporates the needed functionality with a qualitative analysis. Following that application-based, scalable design schemes of the hardware are proposed along with various strategies for the design of a power-efficient DPM-hardware.

5.1. Study of Various Implementation Schemes of the Proposed Approach

To study about the possible implementation schemes, this section presents the various issues that need to be considered from the offline analysis. Details about the data representation and further information that can be obtained from results of offline analysis data and the suitability of software versus hardware implementation methods are presented.

5.1.1. Data obtained from offline analysis:

The methodology proposed for the offline analysis of CFG in the previous chapter provides the following data regarding the dynamic power management of peripherals. This applies for peripherals being accessed in a single thread of embedded application.

- 1) Locations of **Switching Points** in a CFG, for various peripherals accessed.
- 2) Corresponding switching actions to be performed for several devices.
- 3) The specific device identity that needs to be controlled.

- 4) Most suitable power-down state among multiple power-down states depending on the idle time durations for a device and its support for such modes.

Illustration:

In general, the proposed methodology for the offline analysis of CFG (Chapter 4) suits for CFGs at any abstraction level. To discuss about the practical implementations of the proposed approach, the CFG generated at assembly code level seems an effective option to explore and demonstrate. Hence for the rest of the document, the CFG is considered to be generated at assembly code level. Here, every basic block (nodes) of the CFG can be identified with the program counter address of the first instruction in the block.

Figure 5.1: Assembly Code and Equivalent CFG representation with block addresses

As illustrated in a small model CFG of Figure 5.1, every CFG has basic blocks identified by an address location. As a result of the offline analysis, the locations of the *Switching points* are obtained and they are highlighted in the CFG.

As an example, say, the basic block with address 0x0048 is chosen as the switching point location to turn-ON a device numbered as “1” (“A” point). In this scenario, the basic sets of information characterize a switching point with the following items as shown in Figure 5.2:

- ❑ Address Location of the switching point
- ❑ The Device controlled
- ❑ Suggested Switching state

Address Location of the Switching Point	Device Controlled	Suggested Switching State
0x0048	1	ON

Figure 5.2: Basic representation of a switching point data

5.1.2. Switching Points Causing Effective State-Transitions:

As explained earlier, the action of a Switching point basically produces a transition in the State of a peripheral device. In short, a Switch-On point puts a device from a low-power state to an “active” functioning ON state and a Switch-OFF point changes the device from an “active” functioning state to a “Low power” OFF state. The following sub-sections deals with the relationships between switching points and the traversal orders of program execution.

Execution Order of Switching Points:

The data conveyed and the state transitions executed in turn by a *Switching point* are specific to the path along which it gets executed. i.e. It depends on the path along which the execution reaches an I/O node or from which the execution comes out of an I/O node. In other words, the actions (i.e. Switching ON/OFF) suggested by a Switching Point considers the worst-case scenario to ensure not to compromise either the *performance* or *profitability*. To elaborate on this concept, the following examples are provided in view of a device having a single power-down state. i.e. either

ON/OFF. The role of a Switch-ON point and a Switch-OFF point along various paths of program execution is illustrated as follows:

Figure 5.3: Traversal Orders of Switching points during program execution

In figure 5.3,

- The Switch-On point ‘A@q’ ensures performance by Switching-On the device before entering the I/O node along the path “L → p → A@q → r → I/O”.
- Similarly, the Switch-On point ‘A@j’ ensures performance by Switching-On the device before entering the I/O node along the path “L → i → A@j → r → I/O”.

But, when the program execution takes the path “L → i → A@j → A@q → r → I/O”, the Switch-On point ‘A_q’ is unnecessary as the device gets switched ON beforehand at the point ‘A_j’ itself.

Identifying Effective Transitions:

As explained above, as the program execution takes various paths, it encounters several **Switching points** irrespective of the order in which they are traversed along a path. If the switching points along a path cause effective state transitions as often as possible, it aids in improving the overall efficiency. This is because it avoids the extra effort that is needed to be spent on monitoring whether the switching point really causes an effective state-transition or it just leaves the device in the same state. For example, if a Switch-ON point has already caused the necessary state transition of putting the device in the “active” state, the program execution need not encounter another Switch-ON point which performs the same action.

To summarize, the following conclusions can be made to identify effective transitions:

1) For a peripheral device with only ON/OFF states:

For an effective transition while program execution, any order of traversal of switching points should cause only the following transitions:

- i. Transitions from the “active (A)” state to the “Low-power (L)” state.
- ii. Transitions from the “Low-power (L)” state to the “active (A)” state.

2) For a peripheral device with multiple power-down states:

For an effective transition while program execution, any order of traversal of switching points should cause only the following transitions:

- i. Transitions from the “active (A)” state to any of the “Low-power” (i.e. L₁, L₂, etc.) states.
- ii. Transitions from any “Low-power” state to the “active” state.
- iii. Transitions from any “Low-power” state to another relatively “Lower power” state. i.e. From L_n to L_(n-1).

Representation of Effective Transitions in Implementation:

With the above explained methodology to identify effective transitions, a quantitative measure is needed to represent the effective transition to aid for a better implementation strategy. The knowledge of relative locations of **switching points** in the CFG is effective in this aspect. Here a quantity called “*Effective Transition factor*” is defined as follows:

Definition: Effective transition factor is the shortest distance between a switching point of a particular type to the nearest neighbor switching point of another type that can cause an Effective transition in the state of the peripheral device.

Hence, every switching point is assigned an *effective transition factor* based on the shortest distance of the nearest switching point causing an effective transition. For example in Figure 5.3, assume all the blocks are uniform in size and value as 1 unit. Then both the switch-on points “A@q” and “A@j” will have an effective transition factor of 3, to meet the nearest switch-off point “L@s”.

5.1.3. Software and Hardware-based Implementation Schemes:

The features about the data obtained in the offline analysis that are discussed in the preceding sections can be utilized to characterize a switching point in a further detailed way. With the knowledge of the effective transition, a switching point can be characterized with the following information:

1. Location Address of a Switching Point
2. Switching Action
3. The device being controlled
4. Most suitable power-down state suggested for transition
5. Effective transition factor

With this characterization of the switching point, the proposed methodology can be realized practically in two major ways viz., software implementation which can be either dynamic or static and dynamic custom hardware implementation.

Software Implementation Schemes

In a dynamic software based approach, a dedicated program/task can monitor the status of the program execution. This can be done by monitoring the Program counter of the CPU using special instructions or via some profilers which updates a variable regularly when there is a hit in the interested regions of the code. The limitations of this approach are:

- ❑ The execution of the program/task depends on the priority given to this task if it is running on the same processor. In addition to this, it has its own computation complexities for the searching operation in determining the interested regions of the application code. All these incur additional overheads and latency in executing the power transition control words.
- ❑ Secondly, in certain cases, it might involve changing/modifying the application code to incorporate the methodology efficiently. This approach is considered cumbersome and is not recommended in general.

With the information obtained from the Switching points, the meta-data can be fed back into the compiler, making it similar to static power management techniques. This method does not involve dynamic monitoring of the program flow. Hence it may have less overhead than an approach which involves monitoring of program execution. But the disadvantages of this static approach are:

- ❑ Similar to the dynamic software scheme explained above, there is a need for modification and alteration of application code as an intrusion.
- ❑ Less flexibility as there is a need to examine and modify the compiler output every time there is a change in the code.
- ❑ Possibility of increased memory usage for every application.
- ❑ Not possible to make use of the “effective transition” concept discussed in the above section. In other words, there will be more overheads if there is a need to check the current status of the device every time a switching action is performed.

Custom Hardware Implementation Scheme

Another approach is by constructing a lean, power-optimized hardware that can both monitor the status of program execution and other runtime characteristics and can convey appropriate power management decisions to the system. Though this approach has an obvious, inherent cost for the extra area and power consumed by the hardware, this approach can effectively circumvent the issues in the software-oriented approach and still can achieve effective overall energy savings for a range of applications. As part of this research work, the suitable hardware architectures are proposed and analyzed to serve as the Dynamic power manager-hardware (DPM-hardware). The

following section discusses about the required functionalities expected of the hardware and then investigates on a range of architectures in the subsequent sections.

5.2. Constructing a Custom Hardware for DPM

The proposed methodology in the previous chapter can be realized by constructing a lean custom hardware with the very basic functionalities. Those functionalities that are required of the custom hardware are examined as follows. The functions that need to be performed can be categorized broadly as: Monitoring the control flow of program execution and Passing the appropriate decision for power management associated with every switching point as and when they are encountered.

5.2.1. Monitoring the Control Flow of Applications

The application is considered at the control flow level in our methodology. Appropriate blocks of the CFG are chosen for regulating the state of the peripherals for power management. The monitoring method and the points of interest is determined according to the abstraction level at which the CFG is obtained. As mentioned, for the hardware-based approach in our work, we assume that the CFG is obtained at assembly level. Hence monitoring the Program Counter (PC) gives finer information about the current status of the program flow.

In such a control flow graph, every targets of jump and call instruction mark the start of a block in the graph. Hence every block can be identified by the address of the start instruction in that block. When a program counter hits the start address, it implies the execution of the whole block.

At this assembly code level, the execution time of every block having an average of ten instructions is relatively negligible compared with the timing requirements of a peripheral turn-on/off. Hence for all cases, the switching points can be assumed to be at the start of a block unless the block is relatively large or is a loop header. Under the first exceptional case, the larger block can be split at the point necessary. To handle the loop header case, the switching action can be executed after a certain number of iterations with information obtained about the loop bounds.

With such a model, the switching points of interest are identified as the addresses of the blocks in which they get placed in the offline analysis. Hence the basic functionality that is required is to identify those instructions of interest whose addresses match that of switching points. This can be realized by an array of comparators that check the match of the PC values against a list of stored addresses in memory.

5.2.2. Control Words for Power Transitions

Once the monitoring action determines that a switching point is encountered during the dynamic program execution, the DPM unit has to output a control word to put the peripherals in the corresponding states. Every switching point is associated with a switching action and other details as mentioned in the earlier sections. An example to illustrate the control action associated with a switching point is shown as follows:

Let's assume that the 32-bit address location 8080AA90H (hexadecimal) is a switching point which commands a device, here named as 'device_1' to switch to state "S". Then the typical processing that can happen with that address and the associated control word is illustrated with the following pseudo-code:

```
If (PC_Status = 8080AA90H)                (#1)
{
  If current_state (device_1) ≠ 'S' (i.e. the input command) (#2)
  {
    Put Device_1 into 'S' state ;  (#3)
  }
}
```

From the above piece of pseudo-code, it can be inferred that the following items of data need to be represented as control words and given as output of the DPM unit:

1. The device number (among several other devices) that needs to be controlled.
2. The device operating state (e.g. ON/OFF) suggested for transition.
3. A suitable quantitative value that suggests for the next effective transition so as to minimize frequent comparisons of unnecessary transitions. i.e. to enable (#2 of pseudo-code = TRUE), in most cases as possible.

With the aforementioned necessary functionalities and data, the conceptual view of the required hardware architecture is proposed in the next section and then various suitable hardware architectures are investigated for its employability.

5.3. Proposed Architecture for the DPM Hardware

Considering the essential functionalities of the DPM hardware, a basic architectural design is proposed which is elaborated in this section. As discussed in the chapter on literature review, the use of a timer has been in practice since the early 90's for the power management of hard disks [13]. Also, a particular value of a time duration is predicted based on some calculations like usage history, etc. Since from the offline analysis (the proposed methodology in chapter 4), the peripheral accesses and idle time durations are identified, we have the appropriate prediction scheme for power-efficient operation of the peripherals. The hardware-assisted approach presented in this chapter, draws inspiration from these two ideas and proposes an effective methodology for the DPM of embedded applications that gives opportunities for prediction. But under certain cases, the timer becomes optional and only the monitoring hardware is retained as will be explained in the subsequent sections.

Conceptual View of the DPM Hardware:

Basically, the hardware architecture consists of the following important functional units:

- ❑ Memory
- ❑ Monitoring unit
- ❑ Re-loadable Timer unit

The results of the offline analysis of the algorithm are stored in a memory i.e. the address locations of the switching points and the corresponding control actions along with the time duration to quantify the next effective transition are stored in the memory. A hardware unit that compares the incoming program counter value against the stored switching point location is termed as the *Monitoring unit*. Once there is a match, the corresponding power transition action of the switching point matched is given as output along with a value for the *re-loadable timer* to inform about the next use of the monitoring unit along the time line. The conceptual view of the proposed architecture is shown as a block diagram in Figure 5.4. The monitoring unit serves as the heart of the DPM hardware.

Figure 5.4: Conceptual View of the DPM Hardware Architecture

The functional characteristics and the hardware realization details of every unit mentioned above are elaborated in the following sections.

5.3.1. Memory Requirements

To store various necessary data required to incorporate the needed functionality, certain amount of memory is required. The necessary data items are listed out as follows:

Address Locations of Switching Points: To check the match of a program counter value (which is indicative of the execution flow) against a switching point value, the address locations of switching point need to be stored in a block of memory.

For example, consider a 32-bit program counter of any embedded processor. Hence, the code will be executed from a 32-bit instruction memory. Among those several locations, a few locations will be found as the suitable places for locating the control words for switching points from the offline analysis. But all the 32-bits may not be needed to characterize the location of the switching point. In most embedded applications, an address space width of only 24 is used addressing 2^{24} locations. Also, the instruction length of most of the embedded processors is 4 bytes. This means that the last two bits of the locations of the instructions do not differ. Hence, in this case,

minimum of $(24-2=22)$ bits are sufficient to represent the address location of a switching point. Similarly, this number can be found out depending on the processor and the application address space considered.

Control Words:

Power Transition Command:

The new device operating state associated with every switching point need to be represented by certain number of bits which needs additional memory. Let's start with an example of an application accessing a single device having just two states- either ON/OFF. Hence as a minimum requirement, one bit is sufficient to represent these two states. In general for an application accessing a single device having 'm' power-down states, the number of bits necessary can be given by:

$$\text{Number of bits necessary to represent 'm' states} \geq \log_2 m \quad (\text{Eq.})$$

($\log_2 m$ = Logarithm of 'm' to the base of 2)

If there are several devices accessed by an application, then the memory requirements increases accordingly. This is dealt in a later section (). Here, first the necessity of memory to store the power transition i.e. switching state commands is highlighted.

Effective Transition Factor:

Depending upon the type of implementation, the information about effective transition can be utilized to avoid unnecessary power transitions. Referring to the basic architectural view of Figure 5.4, the effective transition factor associated with every switching point can be loaded into the timer whenever a match is encountered. Hence a certain amount of memory is required to store this information. This data along with the power transition command can be collectively called as ***Control Words***.

5.3.2. Monitoring Unit

In our approach, the required task is to monitor the program counter status to check for a match (comparison-operation) against the stored address locations of the switching point. The program counter gets updated at a constant rate in a typical embedded processor. Hence the comparison operation has to be accomplished before the next input arrives. In short, there is a need to compare the input value coming at a particular rate against a list of stored values.

One of the straight-forward hardware realizations to achieve this function is by using a single comparator and shift registers to store the comparison values. But this method has the inherent drawback of poor performance as the worst-case delay to get an output is equal to the number of comparison values which can be very large. Hence the maximum input frequency this can support becomes dependent on the number of stored values which may not be suitable as a generic design.

The functionality similar to our discussion is required in many standard applications such as table-lookup, telecommunications, networking, IP address resolution, etc. To cater all such needs, a commercial device called Content Addressable Memory (CAM) is used in practice. The chief advantage of CAM is its high performance which gives the output in just a single clock cycle. Before describing the proposed architecture of the monitoring unit, the following section gives a brief overview about the CAM and its applicability to our approach.

Content Addressable Memory:

A Content-Addressable Memory (CAM) compares input search data against a table of stored data, and returns the address of the matching data [51]. CAMs have a single clock cycle throughput making them faster than other hardware- and software-based search systems. CAMs can be used in a wide variety of applications requiring high search speeds. These applications include CPU caches, Telecommunications, Networking, ATM Switches, Hough transformation, Huffman coding/decoding, data compression, image coding, etc. The primary commercial application of CAMs today is to classify and forward Internet protocol (IP) packets in network routers. The function of a router is to compare the destination address of a packet to all possible routes, in order to choose the appropriate one. A CAM is a good choice for implementing this lookup operation due to its fast search capability.

Figure 5.5: Conceptual View of CAM

CAMs Vs RAMs

CAMs are an outgrowth of RAM technology. CAM is designed to enhance data retrieval speed from a particular location in a storage array. Instead of using an address to read the data, such as a RAM, the data is supplied as an input to locate the address. CAM determines if the data is found within a storage array. When a match is found, the responder bit associated with the corresponding memory word is set and the address location of the matched word is sent as output.

Figure 5.6: Simple RAM and CAM compared

Data is stored in RAM at a particular location, which is called an address. In RAM, the user supplies the address and gets the data back. The number of address lines limits the depth of a memory using RAM, but the width of the memory can be extended as far as desired. With CAM, the user supplies the data and gets the address back. The CAM searches through the memory in one-clock cycle and returns the address where the data is found. Figure 5.6 compares a typical RAM and CAM operation. The CAM can be preloaded at device startup and rewritten during device operation.

Basic Core of a CAM:

A basic CAM element consists of input data, a storage location (registers), and a comparator. There are several CAM modes: single match, multiple match and ternary CAM. In single match mode, only one location in the storage array contains input data. If the input data can be found in multiple locations within the storage array, it is called multiple match mode. Ternary CAM supports “don’t care” bits as multiple data. The block diagram of a 4 X 8 CAM is shown in Figure 5.7 [56]. From the Match bus shown, the address of the matched item can be decoded if at least one of them goes high.

Figure 5.7: Basic Architecture of a CAM

For our work, as the addresses of switching points are unique, we consider only single match CAM. The basic block diagram representing a CAM is shown in figure 5.8 with the important pins marked. The write-port of the CAM is used to first store the list of values to be compared in its internal memory.

Figure 5.8: Block Diagram of a CAM with Important I/O Pins

The following table 5.1 describes the important input and output signals (pins) that are shown in Figure 5.8. There may be several optional signals in a generic CAM. Here we discuss only those that are relevant to our study.

Signal Name	Direction	Description
ENABLE	Input	Enable: Control Signal used to enable both write and read operation
RD_CLK	Input	Read Clock: All read (to find a match) operations are synchronous to this clock input.
DIN (n:0)	Input	Data In: Input Data to look up from the CAM during read operation. Also, data to be written to CAM during write operation.
WR_ADDR (m:0)	Input	Write Address: The location in which the data is to be written during write operation.
WR_CLK	Input	Write Clock: All write operations are synchronous to this clock input.
MATCH	Output	Match: This signal indicates the existence of matching data inside the CAM.
MATCH_ADDR	Output	Match Address: The CAM address where the matching data resides.

Table 5.1: Core Signals of a CAM

FPGA-based CAM Implementations:

Usually semiconductor companies provide a full custom solution for CAMs. But in recent days, interests are growing towards FPGA (Field Programmable Gate Arrays)-based implementations for several reasons that prove advantageous [55]. The key advantages are listed below:

1. FPGAs are used in embedded systems to cater a wide variety of applications besides table-lookup. The CAM functionality can therefore be integrated with other logic on the same chip.
2. Implementing a CAM in FPGA technology gives us more flexibility and the possibility to add extra features. It is for example possible to add logic to utilize the ‘effective transition’ factor in our approach by incorporating a timer which determines the timings for monitoring to avoid extra overheads.

These can be implemented on FPGAs as well. Details of such design by popular FPGA vendor, Xilinx, are presented below to help in conveying the capability and features of such CAM implementations.

Reference Design	Depth (No.of words)	Width (No. of bits)	CAM Block (bits)	Read		Write (ns)	No. of SRL16E or RAM 16 X 1.	No. of Block RAM	No. of TBUF
				Match (ns)	Encode (ns)				
BlockRAM	32	8	256	4.5	11.5	2X11.5	16	2	-
BlockRAM	128	8	1k	5.5	15	2X15	64	8	32
BlockRAM	256	8	2k	8.5	19	2X19	128	16	64
SRL16	16	16	256	7.5	7.5	16X8.5	64	-	-
SRL16	32	16	512	8	8	16X10	128	-	-
SRL16	128	40	5k	12	12	16X14	1280	-	32
SRL16	256	24	6k	12.5	12.8	16X15	1536	-	64
SelectRAM+ Memory Cell	256	16	4k	16X12	12	12	256	-	-
SelectRAM+ Memory Cell	4096	16	64k	16X20	20	20	4096	-	-

Table 5.2 Design examples of FPGA-CAM implementation of various sizes and performance specifications.

CAMs are implemented in Virtex family devices of Xilinx by using the reprogrammability of the basic LUT [57]. The designs utilize features including fast dedicated carry chains, distributed RAM, built-in shift registers (SRL16E) and Virtex block SelectRAM+ memory. Basically, three different ways are used for the CAM implementation in Xilinx devices. (1) Design with single cycle read but with slow write access (16 clock cycles) using the shift register SRL16. (2) Design optimized for large width and depth but with slow read access (16 clock cycles). This uses the SelectRAM+ memory and the fast carry chain. (3) Design with a single clock read and write but limited to an 8-bit data width per Block SelectRAM+ memory. Table 5.2 lists these design examples for various CAM sizes and performance specifications.

A detailed survey of CAM implementations and techniques to achieve power efficient configurations can be found in [51]-[55]. Based on such schemes and other task-specific considerations, suitable architectures are proposed for the monitoring unit as part of our work and are presented in the following sections.

Proposed Basic Architecture for Monitoring Unit:

Design-1: CAM-based Monitoring Unit

With the essential functionalities required of the monitoring unit of the DPM hardware, a basic architectural design based on CAM is proposed. This is illustrated in Figure 5.9. Here the memory locations inside the CAM store the distinct switching points found in the offline analysis. The RAM chip stores the associated control words for power transition for every switching point. Hence the address location of a switching point inside the CAM and that of the corresponding control word will be the same. Once the match is found inside the CAM, the corresponding address location is sent as output which enables to read the associated control word from the RAM.

With the core signals of the CAM already been explained in Table 5.1, the important signals of the RAM chip associated with the Monitoring unit is explained in Table 5.2.

Figure 5.9: Proposed CAM-based Monitoring Unit

Signal Name	Direction	Description
CLK	Input	Clock: All read/write operations of the RAM are synchronous to this clock input.
CHIP_ENABLE	Input	Chip Enable: Control Signal used to enable the RAM chip for any operation associated with that chip.
READ_ADDR (m:0)	Input	Read Address: The location in which the associated control word for the switching point is stored.
Output (c-1:0)	Output	Control Word Output: The control word stored in the RAM (c-bits wide).

Table 5.3: Core Signals of the Control Word Memory

Advantages:

As this design is directly based on the CAM, it has the chief advantage of being faster and able to produce the comparison output in just a single clock cycle with the least possible delay for a comparison operation. i.e. This architecture can support the least possible delay in getting an output among any other schemes. Overall, the final output which is stored in the RAM can be obtained in just two clock cycles.

Limitations:

This method employs parallel search and serves as a suitable choice when the values are highly uncorrelated in terms of the bit-values. But this parallel nature causes high power consumption. For example, an 18 Mb CAM running at 250 million searches per second consumes 15 W [54]. Hence when the method is applied for a relatively larger number of comparison values, this becomes overkill as this causes proportional increase in the chip area and power consumption of the circuit. This limitation is inherent to any CAM device. Hence a few alternate techniques have been proposed in the literature to reduce the power consumption and increase efficiency. Such techniques make use of correlation among the bit-patterns of the values.

Improved Architectures for Power & Area Efficiency:

Suitable architectures to improve the power and area efficiency of the previous design are proposed in this section. These schemes become effective alternatives when there are large numbers of comparison values and a certain degree of correlation exists in their bit representations.

Before discussing about the architectures, a short discussion about the properties of a typical address space of application codes is presented.

Address Space of Instructions in embedded applications:

For a typical program running as embedded application in a processor, it has its own address space. It executes its own set of instructions which is stored within this space. As such, successive instructions do not differ much in their program counter values unless they take long jumps to other regions of a lengthy application code or to any other application space. But even when such cases happen, for all practical purposes, such jumps will be relatively less in their frequency of occurrence than the former does. Hence the entire address space can be categorized as several zones (or banks) of address spaces.

Figure 5.10: Typical Address Space and Example of possible Switching point locations

Correlation of the MSBs				
4000	5000	600	700	800
0100	0099	00000	01000	000FF
0500	E999	0FFFF	EAAAA	FAAAA
5000		FFFFF		
F000				

Figure 5.11: Correlation of the Switching Point Locations based on MSBs.

The offline analysis with the algorithm gives the suitable locations of Switching points in this address space. Such switching points occur mostly in the vicinity of the peripheral access instructions in this address space. With this scenario, there are greater possibilities for a certain degree of correlation in the switching point address locations.

This is illustrated in Figures 5.10 & 5.11 for a 16-bit address space. As can be observed, similar to the address locations of the instructions in the given space, the switching point locations can also be categorized based on the common MSBs (Most significant bits) of the address values. Once entering a zone of MSBs, the comparison operation performed just for the remaining LSBs (Least Significant bits) prove sufficient to identify a match for the switching point location.

Utilizing this concept, we propose two other architectures having the switching points stored in a certain number of banks.

Design-2: CAM-based Monitoring Unit with Bank-Selection Scheme

The architecture of design-1 explained previously is more of a straight-forward realization of the monitoring unit without considering the nature of the contents which are the switching point locations. Here, we present a suitable architecture for the monitoring unit based on CAMs with bank-selection scheme based on the address space properties as explained above. This scheme is illustrated in Figure 5.12.

Basically, the values to be compared are categorized based on the zones. The address of each zone is stored in a front-end CAM called the 'Parent CAM'. The addresses of the switching points after truncating their zonal address bits are stored in banks of CAMs called the 'child CAMs'.

Here, the operation of child CAMs is dependent upon the output of the parent CAM. Only if the front-end operation is successful, it enables further search in any of the banks based on the decoded address. Then the output is given to the RAM via an encoder stage.

Figure 5.12: Bank-Selection Scheme with CAMs

Advantages:

- ❑ The chief advantage of this approach over the direct design-1 discussed previously is less power consumption. The major comparison operations among several stored values occur only when the front-end search is successful. Again, only one of the banks is operative at a time. Hence, this avoids the power consumption of other circuitry.
- ❑ Another advantage is that the sizes of individual banks (width & depth) are not restricted as every bank of child CAM will have its own comparator circuitry. Hence this architecture can be suitable for applications where the distribution of switching points among various zones is not so uniform. This also enables the use of a Ternary CAM (which can accommodate 'don't care' bits) as the parent CAM without any restrictions as it can support a wide range of bank selection because of the 'don't care' options.

Limitations:

On the contrary to the previously cited advantage, there may be cases where the distribution of switching points among various zones can be almost equal in numbers as they tend to fall around in the vicinity of peripheral access points. Hence with, the

locations peripheral access points as the base address, the offset values can be categorized as zones. Under such cases, the need for separate comparator circuitry for every bank may not be necessary as it consumes silicon area. The proposed architectural design-3 explained below addresses this issue and suggests suitable scheme for area reduction.

Design-3: Bank-Selection Scheme with Common-Comparators

In this scheme, a common array of comparators is used to cater the comparison needs instead of individual child CAMs. Similar to the previous design, this scheme also splits the values to be compared as banks. But instead of banks of CAMs as in design-2, here the values get store in memory cells or registers. A suitable bank-selection logic can be designed which can be a smaller CAM (like the parent CAM in design-2). Figure 5.13 shows such scheme having four banks each having four values to be compared, thus four comparators. Among the “n” incoming bits, certain “s” bits are used for bank selection logic which serves as the multiplexer select lines to select a bank among the other banks.

Figure 5.13: Bank-Selection Scheme with Common-Comparators -Design-3

5.3.3. Re-loadable Timer Unit

From the above description about the monitoring unit, it can be inferred that it serves as the heart of the proposed architecture. It monitors the program execution status to check whether a switching point is encountered and takes the key decision by identifying the suitable power transitions at that point of program execution. The power consumption of this unit can be controlled by making this circuit active only when needed.

The program counter input arrives continuously at a constant frequency, but the identified switching points of interest are distributed randomly among the available address locations. If the monitoring unit checks every program counter value, it results in additional power consumption. As mentioned in the earlier sections, the quantity called ‘effective transition factor’ conveys about the least possible duration between consecutive switching points. Hence the monitoring unit need not check every program counter value. Instead a methodology can be adopted to enable the monitoring unit only at required periods. This can be accomplished by a re-loadable timer as illustrated in Figure 5.14.

Figure 5.14: Timer - Monitoring unit: Interface

Re-loadable Timer is a counter which triggers an interrupt when it reaches a particular count and can be re-loaded after every rollout. Here, this count value is specified by the ‘effective transition factor’. The width of the timer (in bits) can be decided based upon the possible values that can occur as the ‘effective transition factor’. This depends on the application considered as will be explained in the following section 5.4. But as a general case, it needs to be compared with the power consumption characteristics of the monitoring unit. As the design intends to make their operation mutually exclusive with the timer performing for most duration, then the timer should have lesser power consumption than the monitoring unit. This can be achieved for

most general cases as the monitoring unit requires several bits comparison for a number of values in a single clock cycle.

Advantages of the Timer:

The Timer proves advantageous in key aspects as follows:

- ❑ The chief advantage is: a simple power-efficient timer aids in keeping the monitoring unit idle when it is not needed thus saving a considerable amount of energy spent in every comparison operation.
- ❑ In an embedded application, there will be several loops which consume the major portion of the program execution time. Such loops offer a great deal of opportunity for peripheral energy savings. But in certain cases, there may be a need for a peripheral to be in the ON state after the loop execution. When a program execution gets stuck in a loop, the monitoring unit can wake up after a particular number of iterations of the loop which can just cater the turn-ON time of the peripheral. This information can be calibrated and loaded as the 'effective transition factor' in the timer.
- ❑ In general, because of the specific timer reload value, the DPM-hardware is aware of its period of activity. This reduces both unnecessary overheads and power consumption of the hardware circuitry.

Limitations of the Timer:

The use of a timer also has certain limitations which are listed as follows:

- ❑ The timer reload value gives only a measure of the least possible duration of the next arrival of switching point along the program execution. Hence, whenever the program execution takes some other alternate routes of execution, the monitoring unit will be active though unnecessary, till the next switching point which can be considerably farther. If such cases occur frequently during program execution, the purpose of the timer may be less serving.
- ❑ As our approach doesn't consider the occurrence of interrupts, the purpose of the timer again gets less served when the program execution is serving interrupts. This is because the monitoring unit becomes active after the timer rollout unaware of the interrupt.

- Employing a Timer results in additional memory usage as there is a need to store the timer-reload value for every switching point. Again, the timer itself is a hardware implementation which consumes a certain amount of both silicon area and power. All these add to the extra cost of the overall circuitry.

All these limitations are can be suitably handled depending on the embedded applications considered. These can be circumvented by doing a rigorous analysis of the trade-off between energy savings achieved and the energy spent with and without the timer. This has to determined based on the applications under consideration. Hence depending on the application profiles and their needs and characteristics, the timer can be incorporated accordingly.

5.3.4. Overall Examination of the DPM-Hardware Operations

The previous sections 5.3.1.-5.3.3 described about components of the DPM-hardware and their architectural schemes of the proposed custom hardware for DPM. The monitoring unit forms the heart of the DPM-hardware. In this section, we first examine the overall picture about the real-time functioning of the hardware. The overall architectural functioning is illustrated in Figure 5.15. Among the three proposed designs of the monitoring unit, this holistic view of Figure 5.15 is drawn with design-2 as the reference since it is more generic and encompasses the other two.

The hardware operations that occur in the DPM hardware can be seen as pipelined stages. Figure 5.15 shows every stage with a clocked register.

1. **Front-end Monitor:** As described earlier, all the bits of a program counter need not be monitored to compare against the switching point. Selected portion of the program counter word is chosen and among the bits required number of MSBs can be given as the input to the very first stage-known as the front-end monitor unit. This unit may contain either a parent CAM or other type of logic blocks to perform the job of selecting the bank in which the switching point can be found among several banks. This bank selection unit forms the front-end functional block for designs 2 & 3 only as the design-1 has only one master CAM which performs all the functions.
2. **Searching for Match:** This second stage forms the actual search for the switching point within the banks. This stage is enabled only when the front-end monitor output is successful. In other words, only if the MSBs of the input are indicative of prospective switching point zones, the actual search for the switching point gets enabled. This is achieved using the clocked registers 2a, 2b and 2c.
3. **Locating Control Word:** Control words associated with every switching point are stored in the RAM for all designs. Similar to the previous search unit, locating/outputting the control word happens only upon the search for switching points in the previous stage is successful which is given via an encoder/multiplexer in designs 2&3. Registers 3a and 3b helps in enabling this unit.
4. **Control-word output & Timer:** The control word as explained earlier contains data for both power transition commands and the reload value for timer indicating the occurrence for the next possible match. Once the timer rolls-out, it enables the front-end monitor unit to be active.

As can be observed, the functional units that are operative for most of the times are: (1) Front-end monitor and (2) Timer. Hence depending upon the applications, these units have to be optimally designed for minimum power consumption. In light of this, a qualitative analysis is presented in the following section.

5.3.5. Qualitative Analysis

In this section, we present the qualitative analysis of the area, power and timing characteristics of the monitoring unit architectures of the DPM hardware. These analyses give an idea of the static and dynamic power dissipation and help to arrive at the maximum clock frequency the hardware can support in an embedded system.

Control Word Memory:

Every design has the control words stored in RAM or a separate memory unit. For all cases, it needs a single clock cycle to read its output upon the previous stage successful output. The size and operational power of this memory has a linear increase according to the number of items stored and its bit-width.

Monitoring Unit: Area & Power Analysis:

Here we analyze the relative amount of silicon areas consumed by the three design schemes proposed. For power analysis, we consider the ratio of active digital logic in a design according to its functionality.

- **Design-1:** As this design is a direct CAM-based, this consumes a significant silicon amount of area and consumes much power as the entire circuitry operates in a single clock cycle to give the output. As soon as the input arrives, every cell of the memory and the associated comparator circuitry becomes active to find an effective match. Hence, the proportion of the core monitoring unit logic that is active is 1.
- **Design-2:** In this design, we split the CAMs into as many as banks as necessary considering the application features. This requires an additional logic for selecting the bank such as the address decoder. Hence this design consumes slightly more area than design-1. But only one of the banks is active at any time in addition to the front-end monitor and bank selection logic. If “B” is the number of banks of equal sizes, then the proportion of the core monitoring unit logic that is active equals approximately to $1/B$ of the total logic.
- **Design-3:** As this design employs an array of common comparators, it consumes significantly lesser area than other designs. The added components include the multiplexers to select the inputs among the inputs of several banks. The active proportion of logic is the approximately the same as design-2.

Monitoring Unit: Timing Analysis:

- ❑ Design-1 has the least possible delay as it performs the search operation in a parallel manner using the entire resources. But the delay can be significant if the number of values becomes increasingly high as the inbuilt encoder size will increase. Hence with single-clock cycle latency, the output can be obtained.
- ❑ The latency of Design-2 is determined by the critical path in the three stages of pipeline, i.e., the front-end monitor and the address decoder + an individual bank and an encoder stage.
- ❑ Similar to the architecture of Design-2, there are three stages. The critical path is influenced by the delay of front-end bank-selection logic, sum of the single multiplexer and comparator logic, the final encoder stage.

The qualitative analysis is summarized in Table 5.4.

Architecture	Area	Power		Latency	
		Relative Measure	Active components	Clock Cycles	Critical Delay determined by:
Design-1	High	Very High	Entire CAM	1	CAM block based on its depth
Design-2	Very High	Medium	Front-end CAM, decoder, single bank	3	Front-end CAM or Decoder+ One bank or Encoder
Design-3	Medium	Medium	Bank-selection logic, Comparators & Mux	3	Bank-selection logic or Comparators + Mux or Encoder

Table 5.4: Qualitative analysis of core monitoring unit architecture

The following section presents the application-based scalability considerations for the hardware design schemes described above.

5.4. Application-Based Scalable Design Schemes

The constituent functional hardware units of the DPM hardware can be optimized and scaled depending upon the applications they are used for. This section presents the key concepts that can be applied for scalable design/implementation schemes of the constituent functional hardware units. The sizes of comparators and the associated memory elements used are dependent on the correlation of switching point locations. On the other hand, the memory associated with the control word depends on the number of peripherals and associated power down modes. The following sections elaborate on these details.

5.4.1. Correlation of Switching Point locations

Memory for switching point locations

The offline analysis can output any number of switching points for an application depending upon the number of peripheral access locations (application size), their density, number of peripherals etc. Hence, the number of such switching point locations varies depending upon the application. Also, the bit-width used to represent such locations can have various requirements depending on the application.

Once the bit-width of the switching point location chosen for a given processor, it can be further shortened depending upon their correlation. As mentioned previously under section 5.3.2, from the offline analysis the correlation among the switching point locations can be identified with simple analysis of the combination of the bits representing the address locations.

Comparator & Bank Sizes

Further, because of the correlation in the addresses, it influences both the comparator sizes and the number of banks in a bank-selection scheme. Those address locations that refer to these particular zones can be represented as bank addresses. Hence, depending upon the degree of correlation, both the bit-width requirements and the suitable architecture design for the monitoring unit can be determined. This is illustrated in Table 5.5.

Degree of correlation	Bit-width/ Comparator Size	Suitable Architecture for Monitoring unit		
		Design	Number of Banks (Bank Selection Logic delay)	Individual Bank Width (Comparator Size)
Highly correlated	Less	Design-2&3	Large-Medium	Small
Moderately Correlated	Medium	Design-2&3	Small	Medium
Uncorrelated	High	Design-1	-	-

Table 5.5: Choice of architectures based on correlation of addresses

5.4.2. Control Word Memory & Timer Design Schemes

The general format of the control word for an application accessing single peripheral was presented in section. Also, only two possible power-down modes were considered then for a peripheral. Hence two bits are enough to represent a control word command for power transition. In this section, more general representation of the control word for applications handling multiple peripherals and peripherals with any number of power-down modes are explained.

Application accessing multiple peripherals

For applications accessing multiple peripherals, a single switching point may be indicative of a point of control for more than one device. For example, a switching point may require device-1 to be in the sleep-state and device-2 to be in the idle-state and device-3 to be in the ON state. If such information is represented as two bits, the positions of the bits can be used to identify the device controlled whereas the bits convey the switching state. This is shown in Figure 5.16.

	<u>00</u>	<u>01</u>	<u>11</u>
	Sleep	Idle	ON
Device:	(1)	(2)	(3)

Figure 5.16: General Control Word format to control multiple devices.

Further, with our methodology, it can be noted that the state transitions are specific to the path. Hence, when a switching point refers to a transition of a particular device, the other devices may require to be left undisturbed in the previous state itself. This scenario is illustrated in figure 5.17.

Figure 5.17: Multiple sets of switching points falling at the same location

Hence, in general for applications accessing multiple peripherals, in addition to the ON and OFF state, there is a need to convey the ‘leave undisturbed’ state of information. This is represented as the “Null” state. An example format is shown in Figure 5.18 by the bits “10”.

	<u>00</u>	<u>01</u>	<u>11</u>	<u>10</u>
	Sleep	Idle	ON	Null
Device:	(1)	(2)	(3)	(4)

Figure 5.18: Control word format with null state

Bit-width requirements

As can be observed, depending on application's access to the number of devices controlled and their power-down modes, the bit-width requirements can be determined.

As explained under section 5.3.1, in general to represent a device with 'm' states (power-down modes), the number of bits necessary is given by:

Number of bits necessary to represent 'm' states $\geq \log_2 m$

This relation is applicable for an application accessing a single peripheral whereas the "Null" state has to be included for applications with multiple peripherals.

Hence, for the case of applications with multiple peripherals in an application:

Number of bits needed to represent 'm' states of a device $\geq \log_2(m+1)$

With this relation, the total number of bits to represent the general control word can be obtained for all devices and then adding them together. Accordingly, the bit positions have to be given a standard encoding to identify their corresponding devices.

Effective Transition Factor:

As a corollary to the explanation on effective transition factors section 5.1.2, for the case of multiple peripherals, any order of traversal of switching points should cause to change the state of at least one device excluding the "Null" state. This can be determined by simple bit-manipulations between the current control word and with the neighboring control words following the rules as explained in section 5.1.2. The bits of "Null" state if any can be masked. The shortest distance among neighboring control words obeying this rule is taken as the effective transition factor in this case.

Timer Design Schemes

The ranges of values for the 'effective transition factors' can be determined from the offline analysis results depending on the application considered. This determines the width of the timer which in turn influences the size of the control word memory.

Choosing Bit-width of Timer:

The width of the timer is dictated by the highest possible value of the ‘effective transition factor’. But every control word will have its own range of values for the effective transition factor. If the timer-width is chosen based on the highest possible value, every control word has to allocate enough space to accommodate the specified bit-width to maintain uniformity so as to get stored in a RAM or registers.

Such cases may not be of serious concern when the difference between the highest and least values of effective transition is very less. But if there is a wide difference in the ranges of values, then suitable alternate designs of the timer can be considered to save the storage memory. One such suitable alternate scheme is to store the average of all values of effective transition factor.

Scaling down the frequency:

If the application involves several loops making the ranges of effective transition factor very high for most switching points, then a more appropriate approach is to decrease the delay of the timer by scaling down its input clock. Then the effective transition factor stored in the memory will be expressed in terms of this new resolution of the timer. Further, this scheme can have reduced bit-width requirements.

5.4.3. Selecting a Generic DPM-Hardware Model

To cater to several applications in a system, a suitable fixed and generic DPM-hardware model can be selected. This can be done by studying the characteristics of the range of applications under consideration and determining the suitability of a basic architecture design among the three schemes proposed. With this basic architecture at hand, the other factors regarding the size of control word memory and switching point memory can be decided using appropriate guidelines as suggested below.

As a basic strategy, the number of switching points that can be stored can be limited to a certain number depending on the hardware cost. For each application, the significant switching points (as explained in chapter-4) can be made to first accommodate in the available hardware location for they have better opportunities for energy savings.

Control Word Memory:

The control word which gets stored in a separate memory can be chosen to have a constant format as the resource usage of control words will have a linear proportional increase depending on the associated number of switching points, number of peripherals controlled, power-down modes supported and timer value.

Power Transition Commands:

The control word length can be restricted depending on the number of devices the DPM scheme intends to address. Among multiple peripherals present in the system, they can be ordered in terms of their respective power consumption characteristics and their scope of power management. On comparing their energy savings with the hardware cost, certain devices can be neglected to fit in the generic DPM scheme. This saves the monitoring cost and memory due to lesser number of switching points and control words.

Further, for devices with multiple power-down modes, only certain modes can be considered to get accommodated in the generic architecture. Again, this saves resource usage of control words.

Timer:

For the design of a generic architecture, an option for frequency scaling can be provided so that it can cater a wide range of applications. According to the application under control, this option can be calibrated. The bit-width can be chosen as explained under section 5.4.2.

5.5. Summary

In this chapter, we analyzed the various strategies for implementing the proposed methodology of chapter-4 and the inherent issues. Software-based schemes lack the ability to meet suitable performance requirements for monitoring and search operations. We then investigate on a hardware-assisted monitor and control for the DPM with the offline analysis data. We presented suitable architectures for the DPM-hardware to achieve the necessary functionality. Knowledge about the correlation of the switching point address locations can aid in selecting a better power-efficient

Chapter 5. Hardware-Assisted Monitor & Control for DPM

architecture. Scalability of these architectures and qualitative analysis were presented to determine the suitability of architectures according to the applications.

The following chapter delves into the holistic view of the DPM-hardware in an RTOS-based framework. The performance of DPM is analyzed considering all possible overheads in such a system referring to the systematic framework.

Chapter 6. RTOS-Based DPM Framework

In this chapter, the role of the DPM hardware and its position in an RTOS-controlled system is described. Accordingly, an RTOS-based platform and framework for DPM utilizing the profile analysis data has been presented. Performance analysis of the system is presented considering the overheads and suitable techniques to improve the efficiency for achieving effective energy savings are analyzed.

6.1. RTOS-Controlled DPM Platform

In the previous chapters, we explained the proposed methodology for offline analysis of the application profiles to get relevant information to incorporate DPM for peripheral devices. Then, we investigated various implementation schemes and proposed a hardware-based approach and discussed suitable architectures for this DPM-hardware. In this section, we discuss the position of this DPM hardware in a typical embedded system and explain its role necessary to achieve the functionalities. This aids in determining the factors and other considerations that need to be analyzed for the operation-centric view of an efficient DPM-capable RTOS-based framework.

Most embedded systems owing to the complexity of applications and underlying hardware, constrained performance and power budgets, as well as Time-to-market (TTM) and Non-recurring Engineering costs (NRE) issues, require application developers to use runtime support software. This support usually takes the form of a real-time operating system (RTOS) [58] [59].

Core Functionalities of RTOS:

An RTOS provides a number of essential services to an embedded system designer.

- ❑ Most importantly, it serves as an interface between application software and hardware, providing a consistent foundation for application code.
- ❑ An RTOS manages the execution of and interaction between tasks in an application in a timely manner.

- ❑ Manages system resources- The device driver and memory management portions of an RTOS simplify embedded system design by providing the designer with routines to ease the management of hardware resources.
- ❑ In short, it manages the details of task interaction and provides a simplified interface to hardware resources.

Our approach doesn't consider multi-tasking/task switches, but we intend to discuss about the RTOS services in order to envisage a whole system. Relevant to our work, there are RTOS services which can be utilized to get benefited in an overall system. The chief services that can be utilized:

- ❑ In addition to simplifying the use of hardware, the interrupt service routines (ISRs) provided by an RTOS allow hardware to signal an application. Hence a simple mechanism for the DPM-hardware to communicate to the application can be via such RTOS services.
- ❑ The RTOS provides the designer with timer management routines that may be used without detailed knowledge about the timer hardware in the embedded system. Hence if the resolution tick required by the timer is comparable to the one supported by the RTOS of the intended system, the hardware timer can be replaced with this service.

The general structure to incorporate the proposed DPM-hardware in an RTOS-controlled system is shown in Figure 6.1.

Figure 6.1: RTOS-Controlled DPM Platform

RTOS & DPM-Hardware:

- ❑ The DPM-hardware requires the program counter values to know the current status of the program execution flow. This can be achieved by monitoring the address bus or via some special read instructions supported by the processor. The first approach closely resembles a watch-dog processor (WDP) and a sample implementation style can be seen in [60]. Processor like Infineon Tricore supports certain assembly instructions to read the status of program counter supporting the later approach [61].
- ❑ As the DPM-hardware outputs the control word, it can be made to raise an interrupt whenever there is an effective power transition in the states of any peripheral devices. The output which is the power transition command is shown as the “DPM Message” in Figure 6.1. Here, a simple “DPM interface” service can be devised to accomplish this function. This service can then communicate with the device drivers to effect the power transition.
- ❑ As explained in the previous section on RTOS, a suitable RTOS interface can communicate with the RTOS timer service instead of the timer component of the DPM-hardware.
- ❑ Further, the switching point locations that get stored in the DPM-hardware are specific to the applications. As a simple approach, this can be written into the memory once for a fixed application. But as an enhancement, to cater an embedded system running several applications, a suitable mechanism to load every application’s switching point locations automatically to the DPM-hardware can be assigned to the RTOS. This is shown in Figure 6.1 as “Profile Information”. This is basically the results of offline analysis i.e. the switching point locations and the associated control words- also called Profile data. This can be considered by also giving an option for every application either to choose the DPM-hardware support or not.

6.2. Framework for DPM Utilizing Profile Data

In the previous section, we explained the role of the DPM hardware and proposed an RTOS-based platform with DPM support. In this section, we present a systematic framework incorporating various components to realize these strategies. We also present the performance analysis considering the overheads in such a system and techniques to manage them for improving efficiency.

6.2.1. Components of the Framework

As explained in the previous chapters 3 & 4, a control flow profile-based analysis aids in the application-centric realization of an effective DPM strategy. The peripheral features and the application characteristics influence the amount of energy savings that can be achieved using such approach. In addition, depending upon the type of implementation strategy, the overheads incurred needs to be taken into account. With scalable options in the offline analysis results and the implementation approach, overall efficiency can be ensured.

For an RTOS-based DPM platform encompassing the DPM-hardware, the factors and configurations that influence the overall energy-efficiency and the flow in realizing such a system is represented in Figure 6.2. It depicts the systematic framework proposed to achieve DPM supported embedded systems utilizing the knowledge of profile data of the application codes. The components or the stages of the framework are explained as follows:

Figure 6.2: Framework for DPM utilizing Profile Data

The various components/stages in the realization of the framework:

- ❑ From the application's source code and/or the compiler output, a safe control flow profile information can be obtained at desired level of abstraction with proper labels for nodes. For our DPM-hardware approach, we consider CFGs at assembly code level.
- ❑ The inputs for the offline analysis algorithm are in general the CFG data structure and the peripheral features such as the Turn-On and break-even times for various peripherals and for various supported power-down modes.
- ❑ With the knowledge of the target platform and the implementation style, here, the DPM-hardware, the overhead costs of these factors needs to be accounted in the offline algorithm. The outputs of this offline analysis as explained in chapter-4 are the list of locations of switching points, the associated control words and their significance value.
- ❑ To compensate the hardware energy costs, a smaller number of switching points can be chosen based on their significance as explained in chapter-4. With scalable hardware architectures and the short listing of significant switching points, a minimum overall efficiency can be ensured with the DPM-supported system resulting in the power-efficient operation of peripheral devices.

Such an integrated framework providing the DPM-support in an application-centric manner proves to be an efficient approach for embedded systems that run dedicated tasks. Extracting specific characteristics from applications and utilizing them for DPM makes an embedded system more power-efficient. Depending upon the applications, the suitability of various power management policies can be evaluated to be implemented in the framework.

The analysis of significant switching points and the scalability of hardware architectures provide a range of options to ensure a minimum overall efficiency. Further, the framework can be made versatile to handle any power management options for the CPU too. As it can be controlled by the RTOS kernel, it can also help the ACPI to provide a smooth interface between the OS and the hardware.

6.3. Performance Analysis

The performance efficiency of the system described here is determined by the effective energy savings that can be obtained by executing the switching points that are found in the offline analysis. With the use of the DPM hardware for monitoring and commanding for power transitions, evidently, there are certain overhead costs associated while executing the switching points. This section deals with the study of those associated overheads and techniques to improve the overall efficiency of the methodology.

6.3.1. Cost Functions of the Overheads:

The power transition commands perform the role of switching the peripherals ON and to low power modes. These actions are performed to achieve energy savings of the system by the power-efficient operation of the peripherals. But, to get the effective energy savings, the overheads incurred have to be taken into account. The associated cost of the overheads can be categorized as the temporal latency and the energy expended in performing the switching actions.

Irrespective of the type of implementation of the proposed methodology, the CPU of the system incurs certain overheads as it takes a finite number of clock cycles and energy to execute the power transition control words. In addition, if an RTOS kernel, is used to manage the DPM-hardware, it will account for additional overheads. Also, as we discuss here, a hardware-based approach for monitor and control, the overheads associated with the DPM-hardware also needs to be accounted.

Latency:

RTOS: The hardware outputs the power transition control words as and when it senses an effective transition in the states of the peripheral devices controlled. As mentioned in the previous section, a simple mechanism to communicate this event to the system is by giving an “Interrupt” to the RTOS kernel to execute a piece of code called the Interrupt service routine (ISR). In light of that, the latencies associated with the serving of interrupts in an RTOS needs to be taken into account.

Such latencies arise due to the finite amount of time taken for tasks such as saving the CPU's context, execution time for any kernel codes associated with the ISR, device driver, etc. To account the *latency* associated with the *RTOS* for our methodology, we quantify in terms time durations (assuming a single CPU operating at a particular frequency) and denote it as T_{LR} .

CPU: The CPU need to execute a set of instructions to communicate with the driver or any device-control mechanism before accomplishing the power transitions. As this needs a finite number of clock cycles, the *latency* duration associated with *CPU* is denoted as T_{LC} .

DPM-Hardware: As described in chapter 5 about the DPM-hardware operation, the number of clock cycles taken to find a match among the switching points and outputting the associated control word depends on the type of architecture chosen. The *latency* duration associated with the DPM *hardware* can be denoted as T_{LH} .

Energy Expenditure:

The latency described above, relating to the execution of certain codes associated with the *RTOS* or the *CPU* evidently has inherent energy expenditure [62]. Here, the overall *energy* overheads associated with *RTOS/CPU* for the execution of power transition commands is denoted as E_{RC} . Similarly, the *energy* spent for the DPM *hardware* operations can be denoted as E_H .

6.3.2. Improving the Efficiency

With the formulation of overhead costs, suitable techniques to achieve an effective overall energy savings are discussed in this section.

Overhead-Aware Identification of Switching Points

As a first measure, the offline algorithm of the proposed methodology in chapter 4 needs to be revisited to account these overhead considerations.

Revisiting the Offline Algorithm

The important parameters in the offline algorithmic analysis are the Turn-ON time of a peripheral device for the timely operation ensuring performance and the break-even time which aids in ensuring profitable energy savings.

Modified T_{ON} Calculation:

The parameter T_{ON} has to now include the latencies associated in accomplishing a power transition. Hence the new T_{ON} is given as the sum of all latency durations with the original T_{ON} .

This is represented in equation 6.1:

$$T_{ON_NEW} = T_{ON} + T_{LR} + T_{LC} + T_{LH} \quad (6.1)$$

Modified T_{BE} Calculation:

The break-even time concept has been explained in detail in Chapter 3. The calculation of T_{BE} is a measure to indicate that the energy spent will be compensated if the device stays idle for a minimum of T_{BE} .

From equation 3.1,

$$P_w \times T_{BE} = E_o + P_s \times (T_{BE} - T_o)$$

Here P_w , P_s denotes the power consumption of the device in its working and sleeping states. The quantities E_o and T_o represent the energy spent in transitions. To arrive at the new value of T_{BE} to account the overheads, the additional energy spent due to the RTOS/CPU needs to be included as in Equation 6.2.

$$E_{O_NEW} = E_o + E_{RC} \quad (6.2)$$

With this modified value for the transition energy, the equation will give a new value of break-even time as T_{BE_NEW} . But the effective parameter determining the turn-off of a peripheral also needs to include the latency spent before turning it OFF. Hence, similar to the modified T_{ON} calculation, the effective parameter in the place of break-even time also needs to include the latency associated with RTOS/CPU and DPM-hardware. Assuming equal latency spent in either power transitions for simplicity, the effective T_{BE} is given by the equation 6.3.

$$T_{BE_EFF} = T_{BE_NEW} + T_{LR} + T_{LC} + T_{LH} \quad (6.3)$$

Regarding the energy spent in the DPM-hardware, it can be dealt in a different way from the other energy overheads. This is discussed in the following sections.

Analyzing Applications & Offline Results

Having the modified parameters as inputs to the algorithm which considers the possible overheads, the offline analysis can be performed in the usual manner. With the newly obtained outputs i.e. the switching points, further simple analysis can reveal the actual opportunities for energy savings.

The application code being considered and the offline results obtained can be evaluated in the following manner to identify the existing opportunities for energy savings.

Density of I/O access nodes: As explained in chapter-4, the sparseness of the I/O accesses gives a general measure for the possible energy savings. Less dense the I/O accesses are, more the opportunity provided for energy savings. Hence, with the new locations of switching points, suitable analysis techniques can be used to get estimates about the sparseness among various I/O access nodes in the control flow graph.

Key paths of execution: With the standard approach of the offline analysis, the estimates of average energy savings can be obtained by choosing a considerable number of random paths that are indicative of the actual dynamic behavior of the program. Further, the critical paths and the most frequent paths can be identified through profiling runs. This can serve as a more practical indicator of energy savings for applications according to the platform considered.

Customizing DPM-Hardware

- The DPM-hardware explained in chapter-4, is designed to be as lean as possible in terms of area and as less-operative as possible to save power. Most of the times, the effective operating units will be either the timer or the front-end monitoring unit. But still, there will be a non-zero operating power consumption of the custom hardware.
- Performing the power analysis of the DPM hardware architectures, help in determining the effective hardware power consumption. With this result and

the results of the offline estimates, the applicability of this approach for a given application at hand can be analyzed.

- ❑ In short, the energy spent in the hardware should account for a lesser fraction when compared with the energy savings that can be obtained via the DPM of peripherals. Accordingly, the architectures can be scaled to suit the application in an attempt to make the energy savings via the unused peripherals greater. This decision can be further scrutinized by selecting a generic DPM-hardware model to obtain average energy savings of the system as described in section 5.4.3 of chapter-5.

Selection of Significant Switching Points

- ❑ As explained and illustrated in Chapter-4, after the standard phases of the offline analysis, the switching points can be categorized based on their estimated significance. Hence, selecting only a minimum number of switching points based on the DPM-hardware architecture in hand, can serve as an alternative or sometimes complementary approach to the above mentioned strategy.
- ❑ This approach can be further improved by making more accurate estimates of the significance of Switch-OFF points which can be determined by profiling runs of the application.

6.4. Summary

In this chapter, we proposed an RTOS-based DPM platform encompassing the DPM-hardware in a system supporting runtime operations. After explaining about the key interactions between them, we presented a systematic framework for DPM utilizing the profile-analysis data. Further, we presented the performance analysis of the system considering the various factors influencing the energy efficiency and provided suitable techniques to improve the overall efficiency. Figure 6.2 summarizes the various stages of this systematic framework for application-centric DPM in embedded systems.

In the next chapter, we summarize and conclude this thesis work and outline key recommendations for future work.

Chapter 7. Conclusions and Future Work

This final chapter summarizes the contents of the thesis highlighting the salient factors and presents the future recommendations that can be addressed to improve upon the research work presented in this thesis.

7.1. Conclusion

In this thesis, we have successfully addressed the importance of application profile-based approach for the dynamic power management of the peripherals in an embedded system. We have proposed profile based techniques to incorporate both performance-centric and power-efficient operation of the I/O devices.

With the proposed methodology in this project, we have formulated an effective strategy to find out the opportunities for the energy savings of peripheral devices in an embedded system. The algorithm devised for offline analysis finds suitable locations in the code to switch the device into appropriate power modes. It lends well for applications accessing both single and multiple independent peripherals. Further enhancements were made to cater devices supporting multiple power-down modes.

The results of our detailed simulation analysis of the algorithm reveal that the amount of energy savings achieved has a proportional increase depending on the sparseness of I/O accesses. Also, the energy efficiency can be further improved in real scenarios where the knowledge of the criticality/frequency of paths taken during program execution can be determined using offline profiling runs with suitable inputs. Further, the algorithm also attempts to minimize the overheads by suggesting the order of significance for every switching point that can put the device into low power modes. Based on the significance value, required number of switching points can be selected.

A custom hardware to implement this methodology is designed to perform the role as the “Hardware Dynamic power manager”. In addition to the area, power and timing analysis of the hardware, correlation among switching-point locations have been shown to influence the eventual architecture. In our design, the timer and the front-end monitoring unit determine the effective hardware energy costs, thereby revealing

where the design efforts for power optimization has to be focused. Further, techniques for accommodating a range of applications with the help of a fixed architecture have been proposed.

Finally, an RTOS-based framework encompassing these techniques has been proposed along with techniques to customize for energy efficiency. This includes the investigation of factors influencing the performance of DPM strategies in the system such as the overheads of the CPU/RTOS and customized DPM-hardware.

7.2. Scope for Future Work

This work has laid the foundation for the development of an integrated framework for the application-centric power management in an embedded system. Subsequent to this work, there is clearly a need for numerous other issues to be addressed. Some of the major items of future work are proposed as follows:

Application to CFG Conversion: In this research work, the control flow representation of application code is of central concern. Though there are commercial tools available to generate safe CFGs, it will be desirable to have an appropriate methodology depending on the applications and platform targeted. Hence, techniques to generate exhaustive and safe control flow graphs of the code at the desired level of abstraction and to provide suitable standards for specifying address locations of the switching points become a necessary work package to be addressed. The compiler can aid in the generation of CFGs as well as to provide support for issues like irreducible loop structures, loop unrolling and other compiler optimization effects.

Automating offline profile analysis: The algorithmic analysis done can be integrated as part of the compilation process. This requires appropriate techniques for the identification of peripheral access blocks in the CFG and inputs such as the associated peripheral features. Techniques like feeding the analyzed data back into the compiled output, similar to a static approach for power management can be also considered as an alternative approach. This may have less monitoring overhead to infer the current execution status. But, rigorous analysis of all other overheads has to be done to evaluate the overall profitability of this alternate approach.

Dealing with Interrupts & Multi-Tasking

The proposed methodology needs further enhancements and evaluation to handle task switches and interrupts. The focus of our work has been on stand-alone applications with peripheral accesses as is the case with most of the profile based power management techniques in the literature. It is worth investigating the implications on the overall efficiency of the power management scheme in the presence external interrupts. Extending this approach to multi-tasking systems will also be of interest. It is envisaged that inter-dependence and inter-communication will play a significant role in such scenarios.

Integrated Dynamic Power Management for CPU:

In our research work, we have dealt with the DPM of peripherals based on the control flow profile of the applications. As mentioned in the chapter on literature review, several works have been reported for the dynamic voltage/frequency scaling of the processors for their power-efficiency. Similar to our approach, certain locations of the CFG can be chosen to scale the voltage/frequency depending on the task deadlines and other issues. As our approach assumes a single frequency of operation, the interplay between the proposed DPM and dynamic voltage/frequency scaling can lead to interesting possibilities.

Power-Efficient and Versatile DPM-Hardware:

Various power-saving schemes of CAM implementation such as using a segmented approach for comparison, employing certain pre-computation and encoding of the stored data, etc. can be considered to enable the DPM-hardware more power-efficient. While FPGA-based implementations offer more flexibility, the hardware can also be investigated for a full custom solution in ASIC for better power efficiency. Also, the employability of such DPM-hardware can be investigated for the dynamic voltage/frequency scaling of processors where it can perform the function of giving commands at specific locations of the CFG to scale the frequency of the CPU.

Evaluation in an ACPI/RTOS Environment

With the RTOS-based framework encompassing the DPM-hardware, can be further analyzed to consider the effects of RTOS and its runtime characteristics and evaluated with suitable benchmarks. In addition, further research can be done to evaluate the

Chapter 7. Conclusions and Future Work

support for ACPI. For devices with multiple power-down modes, the ACPI support in an Operating system will provide a smooth interface for power transitions. In this context, suitable methodology and framework for dynamic power management that are appropriate for incorporation into ACPI supported RTOS environment can be devised.

Bibliography

- [1] L. Benini, A. Bogliolo and G. D. Micheli, "System-level Dynamic Power Management", *Proc. of the IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, pp.23, 1999.
- [2] L. Benini, A. Bogliolo and G. D. Micheli, "A Survey of Design Techniques for System-Level Dynamic Power Management", *IEEE Trans. VLSI Systems*, vol. 8, no. 3, pp. 299-316, June 2000.
- [3] Y.H. Lu, "Comparing System Level Power Management Policies", *IEEE Design and Test of Computers*, vol. 18, pp. 10 – 19, Mar. 2001.
- [4] P.J. Koopman, "Embedded Systems Design Issues (The Rest of the Story)", *Proceedings of the International Conference on Computer Design*, pp.310, 1996.
- [5] C.Brake, "Power Management in Portable ARM based Systems", *Online*: <http://www.windowsfordevices.com/articles/AT2608778568.html>
- [6] Intel, Microsoft, and Toshiba. "Advanced configuration and power interface specification", *Online*: <http://www.acpi.info/>
- [7] D. Shin and J. Kim, "Optimizing Intratask Voltage Scheduling Using Profile and Data-Flow Information", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 369-385, Feb. 2007.
- [8] T. Okuma and T. Ishihara, "Software Energy Reduction Techniques for Variable-Voltage Processors", *IEEE Design and Test of Computers*, vol. 18, no. 2, pp. 31-41, Mar-Apr. 2001.
- [9] D. Shin and S. Lee, "Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications", *IEEE Design and Test of Computers*, vol. 18, no. 2, pp 20-30, Mar-Apr. 2001.
- [10] N.AbouGhazaleh, D.Mosse, B.Childers, R. Melhem and M.Craven, "Collaborative Operating System and Compiler Power Management for Real-Time Applications", *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pp.82-115, 2003.
- [11] A. Azevedo, R. Cornea and I. Issenin, "Architectural and Compiler Strategies for Dynamic Power Management in the COPPER Project", *Proc. of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp.25, 2001.

Bibliography

- [12] L.Niu and G.Quan “Peripheral-Conscious Scheduling on Energy Minimization for Weakly Hard Real-time Systems”, *Design, Automation & Test in Europe Conference & Exhibition*, pp.1-6, 2007
- [13] Dell Computer Corp., “Dell System 320SLi User’s Guide”, June 1992.
- [14] Hewlett-Packard, “Kittyhawk Power Management Modes”, Apr.1993.
- [15] F. Dougllis, P. Krishnan and B. Bershad, “Adaptive Disk Spin-Down Policies for Mobile Computers,” *Proc. Second USENIX Symposium. Mobile and Location-Independent Computing*, pp. 121-137, April 1995.
- [16] M.B. Srivastava, A.P. Chandrakasan and R.W. Brodersen, “Predictive System Shutdown and Other Architecture Techniques for Energy Efficient Programmable Computation,” *IEEE Trans. VLSI Systems*, vol. 4, no. 1, pp. 42-55, Mar. 1996.
- [17] E.Y. Chung, L. Benini and G.D. Micheli, “Dynamic Power Management Using Adaptive Learning Tree,” *Proc. Int’l Conf. Computer-Aided Design*, pp. 274-279, Nov. 1999.
- [18] C.H. Hwang and A.C. Wu, “A predictive System Shutdown Method for Energy Saving of Event Driven Computation”, *Proc. of Intl. Conference on Computer-Aided Design*, pp.28-32, 1997.
- [19] L. Benini, A. Bogliolo, G.A. Paleologo and G.D. Micheli, “Policy Optimization for Dynamic Power Management,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18,no. 6, pp. 813-833, June 1999.
- [20] Q. Qiu and M. Pedram, “Dynamic Power Management Based on Continuous-Time Markov Decision Processes,” *Proc. of Design Automation Conference*, pp. 555-561, June 1999.
- [21] T. Simunic, L. Benini, P. Glynn and G.D. Micheli, “Dynamic Power Management for Portable Systems”, *Proc. of Intl. Conference on Mobile Computing and Networking*, pp.11-19, Aug. 2000.
- [22] E.Y. Chung, L. Benini, A. Bogliolo, Y.H. Lu and G.D. Micheli, “Dynamic Power Management for Nonstationary Service Requests”, *IEEE Transactions on Computers*, vol. 51, no. 11, pp. 1345-1361, Nov. 2002.
- [23] D.Li, P.Chou and N.Bagerzadeh, “Mode Selection and Mode-Dependency Modeling for Power-Aware Embedded Systems”, *Proc. of Asia South Pacific Design Automation Conference*, pp.697-704, 2002.

Bibliography

-
- [24] V. Swaminathan and K. Chakrabarty, "Energy Conscious, Deterministic I/O Device Scheduling in Hard Real-Time Systems", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 22, no. 7, pp.847-858, Jul. 2003.
 - [25] H. Cheng and S. Goddard, "Online Energy-Aware I/O Device Scheduling for Hard Real-Time Systems", *Proceedings of Design, Automation and Test in Europe*, pp.1055-1066, Mar. 2006.
 - [26] L. Benini, A.Bogliolo, S.Cavallucci and B.Ricco, "Monitoring System Activity for OS-Directed Dynamic Power Management", *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pp.185-190, Aug. 1998.
 - [27] Y.H. Lu, T. Simunic and G. D. Micheli, "Software Controlled Power Management", *Proceedings of the Seventh International Workshop on HW/SW Codesign*, pp. 157-161, 1999.
 - [28] Y.H. Lu, L. Benini and G.D. Micheli, "Operating-System Directed Power Reduction", *Proceedings of 2000 International Symposium on Low Power Electronics and Design*, pp.37-42, 2000.
 - [29] Y.H. Lu, L. Benini and G.D. Micheli, "Power-Aware Operating Systems for Interactive Systems", *IEEE Transactions on VLSI Systems*, vol. 10, no. 2, pp.119-134, April 2002.
 - [30] Y.H. Lu, L. Benini and Micheli G D, "Requester-Aware Power Reduction", *Proceedings of 13th Symposium on System Synthesis*, pp.18-24, Sep. 2000.
 - [31] C. Gniady, A.R. Butt, Y.C. Hu and Y.H. Lu. "Program Counter Based Prediction Techniques for Dynamic Power Management", *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 641–658, June 2006.
 - [32] C.Isci, G. Contreras and M. Martonosi, "Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management", *Proceedings of the 39th Annual International Symposium on Micro-architecture*, pp.359-370, 2006.
 - [33] X. Liu, P.Shenoy and M.Corner, "Chameleon: Application Level Power Management with Performance Isolation", *IEEE Transactions on Mobile Computing*, Vol. 7, Issue 8, pp. 995 – 1010, Aug. 2008
 - [34] T.Heath, E.Pinheir, J.Hom, U.Kremer and R.Bianchini, "Code Transformations for Energy-Efficient Device Management", *IEEE Transactions on Computers*, Vol. 53, No. 8, pp. 974-987, Aug. 2004.
-

Bibliography

- [35] C. Im, H. Kim and S. Ha, "Dynamic Voltage Scheduling Technique for Low-Power Multimedia Applications Using Buffers," *Proc. Intl. Symposium on Low Power Electronics and Design*, pp. 34-39, 2001.
- [36] D. Ramanathan, S. Irani and R. Gupta, "An Analysis of System Level Power Management Algorithms and Their Effects on Latency", *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 21, No. 3, pp.291-305, Mar. 2002.
- [37] S. Irani, G. Singh, S.K. Shukla and R. Gupta, "An Overview of the Competitive and Adversarial Approaches to Designing Dynamic Power Management Studies", *IEEE Transactions on VLSI Systems*, Vol. 13, No. 12, 2005.
- [38] O.S. Unsal and I. Koren, "System-Level Power-Aware Design Techniques in Real-Time Systems", *Proceedings of the IEEE*, Vol. 91, No.7, pp. 1055, 2000.
- [39] Trimaran- Compiler Infrastructure. Online: www.trimaran.org/
- [40] Suif-Infrastructure. Online: <http://suif.stanford.edu/suif/>
- [41] T.Ball, "The Use of Control Flow and Control Dependence in Software Tools", *PhD Thesis*, University of Wisconsin-Madison, August 1993.
- [42] K.Cooper,T.Harvey and T.Waterman. "Building a Control-flow Graph from Scheduled Assembly Code", *Technical Report*, Department of Computer Science, Rice University, June 2002.
- [43] A.V. Aho, R. Sethi and J.D. Ullman. "Compilers Principles, Techniques and Tools", *Addison-Wesley Publishers*, 2006.
- [44] D. Kastner and S. Wilhelm, "Generic Control Flow Reconstruction from Assembly Code", *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, pp. 46-55, 2002.
- [45] WCET analysis tool- Online <http://www.absint.com/products.htm>
- [46] C.Ferdinand and R.Heckmann, "Worst-Case Execution Time Prediction by Static Program Analysis", *Proc. of the 18th Intl. Sym. on Par. and Dist. Processing*, pp.125, Apr. 2004.
- [47] H. Theiling, "Control Flow Graphs for Real-Time System Analysis - Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis", *Doctoral Dissertation*, Saarland University, 2002.

Bibliography

- [48] A. Azevedo, I. Issenin and R. Cornea, "Profile-based Dynamic Voltage Scheduling using Program Checkpoints", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp.168-175, 2002.
- [49] S. Talli, R. Srinivasan and J. Cook, "Compiler-Directed Functional Unit Shutdown for Microarchitecture Power Optimization", *Proc. of the IEEE Intl. Performance, Computing and Communications Conference*, pp. 372, 2007.
- [50] A.A. Khan and H. Singh, "Petri Net Approach to Enumerate all Simple Paths in a Graph", *Electronic Letters*, Vol. 16, No.8, pp. 291-292, Apr 1980.
- [51] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey", *IEEE Journal of Solid-State Circuits*, Vol. 41, No.3, Mar 2006.
- [52] K. Pagiamtzis and A. Sheikholeslami, "A Low-Power Content-Addressable Memory (CAM) Using Pipelined Hierarchical Search Scheme", *IEEE Journal of Solid-State Circuits*, Vol. 39, No.9, Sep 2004.
- [53] S. Baeg, "Low-Power Ternary Content-Addressable Memory Design Using a Segmented Match Line", *IEEE Transactions on Circuits and Systems –I: Regular Papers*, Vol. 55, No. 6, July 2008.
- [54] N. Mohan, W. Fung, D. Wright and M. Sachdev, "Design Techniques and Test Methodology for Low-Power TCAMs", *IEEE Transactions on VLSI Systems*, Vol. 14, No. 6, Jun 2006.
- [55] S.A. Guccione, D. Levi and D. Downs, "A Reconfigurable Content Addressable Memory", *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pp.882-889, 2000.
- [56] Actel, "Content Addressable Memory (CAM) in Actel Devices", *Application Note, AC194*, 2003.
- [57] Xilinx, "An Overview of Multiple CAM Designs in Virtex Family Devices", *Application Note, XAPP 201*, 1999.
- [58] Q. Li, "Real Time Concepts for Embedded Systems", *CMP Books*, 2003.
- [59] J.J. Labrosse, "MicroC/OS-II: The Real Time Kernel", *CMP Books*, 2002.
- [60] G.C. Giaconia, D. Stefano and G. Capponi, "FPGA-based Concurrent Watchdog for Real-Time Control Systems", *Electronics Letters*, Vol.39, pp.769-770, May 2003.
- [61] Infineon TC1775, *User's Manual*, V2.0, Feb. 2001.

Bibliography

- [62] R.P. Dick, G. Lakshminarayanan, A. Raghunathan and N.K. Jha, "Analysis of Power Dissipation in Embedded Systems Using Real-Time Operating Systems" *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 22, No.5, pp.615-627, May 2003.

Appendix

Simulation Analysis of Control Flow Graphs with the obtained outputs of
“Switching points” (Refer Chapter-4)

1. Basic Algorithm (Single Peripheral) (A.1-A.3)
2. Case of Multiple Peripherals (A.4-A.5)
3. Case of Peripherals with Multiple Power-Down States (A.6-A.7)

Figure A.1: Basic Algorithm Simulation-CFG with "A" Points Identified

Figure A.3: Basic Algorithm Simulation –After elimination phase (final list)

Figure A.4: Simulation Analysis for Multiple I/O-1

Figure A.5: Simulation Analysis for Multiple I/O-2 (Superimposed with previous I/O-1 case)

Figure A.6: Simulation analysis for multiple power-down modes (mode-2)

Figure A.7: Simulation analysis for multiple power-down modes (mode-1)-Superimposed

Publication resulting from this work

International Conference

- C. Magesh Kumar, M.Sindhwani and T.Srikanthan, "Profile-based technique for dynamic power management in embedded systems", *Proceedings of International Conference on Electronic Design*, pp. 1-6, Penang, Malaysia, December 2008.