OPTIMIZING PROCESSOR AND MEMORY FOR GREEN COMPUTING

by

Mingsong Bi

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCES

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2011
As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Mingsong Bi entitled Optimizing Processor and Memory for Green Computing and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Christopher Gniady  
Date: 9 Dec 2011

John Hartman  
Date: 9 Dec 2011

David Lowenthal  
Date: 9 Dec 2011

Beichuan Zhang  
Date: 9 Dec 2011

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College. I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Christopher Gniady  
Date: 9 Dec 2011
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. This work is licensed under the Creative Commons Attribution-No Derivative Works 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

SIGNED: Mingsong Bi
ACKNOWLEDGEMENTS

Though only my name appears on the cover of this dissertation, a great many people have contributed to its production. I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever.

First, my sincere thanks is to my advisor, Dr. Chris Gniady. I have been amazingly fortunate to have an advisor whose patience and support helped me overcome many crisis situations and finish this dissertation. Your advice and insight were invaluable to the success this work has enjoyed. Thank you for your guidance and help in reaching this important milestone.

A heartfelt thanks to my brother Gen Lu. As a Wildcat’s greatest fan, your passion, soul and knowledge lightened me. Our glorious times in the McKale Center and Arizona Stadium, as well as Recreation Center were such an important outlet throughout the challenges of the recent years. I feel truly fortunate to have a friend sharing so much with each other. Your support over my back cheered me up during the plain and sometimes hard times. I enjoyed and cherished the moment with you on and off court!

Many wholehearted friends have helped me stay sane through these years. Sincere thanks to Xiaofei Qu, Cheng Yi, Fengqiong Huang. Thank you for your accompany with me, sharing our laughters, happiness and even hardships. Your support and care helped me overcome setbacks and stay focused on my graduate study. I greatly value your friendship and I deeply appreciate your belief in me. You are not just friends to me, but also gives me a family feeling in the darkest of times.

Many thanks to my colleagues: Igor Crk, Lei Ye, and Ran Duan. Your wise, cooperation and dedication these years assisted me greatly a to finish my dissertation. Besides, many thanks to my friends around and outside the Department of Computer Science: Yi Lu, Deqiang Mao, Qijing Li, Rui Zhang, Varun Khare, Kyri Pavlou and Somu Perianayagam.

Finally and most importantly, none of this would have been possible without the love and patience of my family. An ineffable grateful thanks will never be enough to satisfy the debt of gratitude I have to them. As a forever soul mate, Shuhan has been a constant source of love, concern, support and strength all these years. My parents, raising and supporting me all the times, also aided and encouraged me throughout this endeavor. I truly and deeply appreciate their generosity and understanding. I would like to express my heart-felt gratitude to them.
DEDICATION

To Shuhan
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>11</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>12</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Energy Management for Server Systems</td>
<td>14</td>
</tr>
<tr>
<td>1.2 Energy Management for Interactive Systems</td>
<td>16</td>
</tr>
<tr>
<td>1.2.1 Processors</td>
<td>17</td>
</tr>
<tr>
<td>1.2.2 Memory</td>
<td>18</td>
</tr>
<tr>
<td>CHAPTER 2 BACKGROUND</td>
<td>20</td>
</tr>
<tr>
<td>2.1 Energy and Power</td>
<td>20</td>
</tr>
<tr>
<td>2.1.1 Dynamic Power and Static Power</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Processor Power States</td>
<td>22</td>
</tr>
<tr>
<td>2.3 CPU Power Management</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Interval-Based Approaches</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2 Task-Based Approaches</td>
<td>24</td>
</tr>
<tr>
<td>2.4 Memory Power States</td>
<td>26</td>
</tr>
<tr>
<td>2.5 Memory Power Management</td>
<td>28</td>
</tr>
<tr>
<td>2.5.1 Hardware Approaches</td>
<td>28</td>
</tr>
<tr>
<td>2.5.2 Software Approaches</td>
<td>29</td>
</tr>
<tr>
<td>2.6 Workloads Characteristics</td>
<td>31</td>
</tr>
<tr>
<td>2.6.1 Non-Interactive Applications</td>
<td>32</td>
</tr>
<tr>
<td>2.6.2 Interactive Applications</td>
<td>33</td>
</tr>
<tr>
<td>CHAPTER 3 DELAY-HIDING ENERGY MANAGEMENT MECHANISMS</td>
<td>35</td>
</tr>
<tr>
<td>FOR DRAM</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Design</td>
<td>36</td>
</tr>
<tr>
<td>3.1.1 Exploiting File-I/O System Calls</td>
<td>36</td>
</tr>
<tr>
<td>3.1.2 All-Ranks-On for Delay Reduction</td>
<td>38</td>
</tr>
<tr>
<td>3.1.3 Predicting Individual Ranks</td>
<td>39</td>
</tr>
<tr>
<td>3.1.4 Increasing Prediction Coverage</td>
<td>40</td>
</tr>
<tr>
<td>3.1.5 Improving Energy Efficiency</td>
<td>42</td>
</tr>
<tr>
<td>3.2 Methodology</td>
<td>43</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS – Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Evaluation</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>3.3.1 Accuracy of Predictors</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>3.3.2 File-I/O Time</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>3.3.3 File-I/O Energy Consumption</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>3.3.4 Energy Reduction with Early-Turnoff</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>3.3.5 Larger Memory Systems</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.1</td>
<td>IADVS design</td>
</tr>
<tr>
<td></td>
<td>4.1.1 Distinguishing UI Interactions</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>4.1.2 Detecting UI-triggered Tasks</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>4.1.3 Correlation Mechanisms</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>4.1.4 Predicting CPU Power Modes</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>4.1.5 Managing Multi-core CPUs</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>4.2</td>
<td>Methodology</td>
</tr>
<tr>
<td></td>
<td>4.2.1 Processor Model</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>4.3</td>
<td>Evaluation</td>
</tr>
<tr>
<td></td>
<td>4.3.1 Accuracy</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>4.3.2 Delay</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>4.3.3 Energy</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>4.3.4 Energy-Delay$^2$ Product</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>4.3.5 The Need for FFRCPU</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>4.4</td>
<td>Implementation</td>
</tr>
<tr>
<td></td>
<td>4.4.1 Implementation Details</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>4.4.2 Results</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>5.1</td>
<td>IAMEM design</td>
</tr>
<tr>
<td></td>
<td>5.1.1 Memory Space</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>5.1.2 UI and Task Correlation</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>5.1.3 Power State Prediction</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>5.1.4 Improving Prediction Granularity</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>5.1.5 Improving Monitoring Accuracy</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>5.1.6 Preserving Performance</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>Methodology</td>
</tr>
<tr>
<td></td>
<td>5.2.1 Performance Demand of Applications</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>Evaluation</td>
</tr>
<tr>
<td></td>
<td>5.3.1 Energy</td>
<td>87</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.3.2 Performance</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>5.3.3 The Need for Dual Prediction</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>5.3.4 Delay Reduction with Early-Detect</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>CHAPTER 6 CONCLUSIONS</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Computer system organization and optimization levels. .................. 14
2.1 Power state transitions and latencies for a Micron 1Gbit DDR2-800 device. ................................................................. 27
2.2 Power specifications for a rank consisting of 8 Micron 1Gbit DDR3-1066 devices. .......................................................... 28
2.3 Architecture of Core i CPU. .................................................. 30
2.4 Memory profile for I/O bound workloads running in a system with 8 GB RAM ................................................................. 33
2.5 Cumulative distribution of processing times of interactive tasks. .... 34
3.1 Anatomy of File-I/O operations. ........................................... 37
3.2 Data structure (shaded area) added to the file table for proposed mechanisms. ................................................................. 40
3.3 Access frequency of all files based on the file distribution over ranks. 44
3.4 Prediction accuracy of the proposed mechanisms. ....................... 45
3.5 Breakdown of file-I/O time. .................................................. 47
3.6 Breakdown of memory energy consumption. .............................. 48
3.7 Energy consumption before and after early-turnoff. ...................... 50
3.8 Results of various predictors under different rank size for TPC-C. .... 53
4.1 GUI menus in GIMP. A: Menu and toolbar area are distinguishable. B: Individual elements are distinguishable. ......................... 55
4.2 IADVS design architecture. .................................................. 56
4.3 Task distribution based on the tasks’ optimal frequency for the FFR-CPU. ................................................................. 64
4.4 Prediction distance from ORACLE’s optimal frequency. ............... 64
4.5 Absolute prediction error. .................................................... 65
4.6 Runtime normalized to ORACLE. ........................................... 66
4.7 Energy normalized to ORACLE. ............................................. 68
4.8 Energy-delay$^2$ product normalized to ORACLE. ....................... 69
4.9 Energy-delay$^2$ product for AMD CPU normalized to ORACLE in FFRCPU. ................................................................. 70
4.10 Measurement setup for CPU power consumption. ...................... 71
4.11 Power consumption in a selected 3-second period during the execution of Scigraph under different DVS mechanisms. ............... 73
LIST OF FIGURES – Continued

5.1 Distribution of tasks with memory in high and low power states. Shaded bar represents the fraction of short tasks extended over 50ms, while shaded-crossed bar represents the fraction of long tasks extended by more than 50ms. .................................................. 75
5.2 OS view of memory space. .......................................................... 77
5.3 IAMEM design architecture. ...................................................... 78
5.4 The power-state prediction algorithm for an upcoming task. \( L_{pd} \) and \( L_{sr} \) are the transition latencies from PD and SR state to PRE state, respectively. .................................................. 79
5.5 The dual prediction algorithm. .................................................. 81
5.6 Task performance demand based on the ORACLE’s optimal power. 86
5.7 Average per-task memory energy consumption for processing each task normalized to ORACLE. ............................................. 88
5.8 Average per-task length normalized to ORACLE. ...................... 90
5.9 Energy consumption of IAMEM with single and dual prediction. 91
LIST OF TABLES

1.1 Power consumption breakdown for IBM p670 eServers with different configurations. ................................. 15

2.1 The P-states of a Phenom II X4 940 CPU with the associated core voltages (VDD) and current operating frequencies (COF). ............ 22
2.2 The C-states supported by a AMD Phenom II X4 940 CPU with associated transition latencies and the state descriptions. ............ 23
2.3 Power specifications for a rank consisting of 8 Micron 1Gbit DDR2-800 devices. .................................................. 27

3.1 Applications and trace statistics. ............................................. 43

4.1 Statistics of CPU demand (Million Cycles) for mouse click events in GIMP. ............................................................... 54
4.2 Mean Coefficient of Variation (CV) of the tasks’ CPU demand. .... 56
4.3 Application trace statistics. ................................................... 62
4.4 Phenom II X4 940 p-states with the associated CPU voltages (VDD), current operating frequencies (COF) and the power consumptions (in Watts). .................................................. 72
4.5 Energy consumed by the CPU. .............................................. 73

5.1 Statistics of application traces. ............................................. 83
5.2 Total memory energy consumption (in Joules) during the entire application runtime. ........................................ 88
5.3 Average delays of the delayed long running tasks in standard IAMEM with the early-detect optimization and alternative design without the optimization. ........................................ 92
ABSTRACT

Energy efficiency has become one of the most important factors in the development of computer systems. Increasingly power-hungry processors and memory subsystem have reinforced the need for aggressive power management. Dynamic voltage scaling has become a common consideration for designing energy efficient CPUs in systems ranging from portable devices to large-scale systems. As applications become more data centric and put more pressure on memory subsystem, managing energy consumption of main memory is also becoming critical. Subsequently in this dissertation, we address the issues in designing energy efficient CPU and memory for personal computing devices as well as large-scale systems.

For large-scale systems, we address memory subsystem dedicated to buffer cache which accounts for the majority of memory usage in server environment. We take advantage of the I/O handling routines in the OS kernel to hide the delay incurred by the memory state transition so that performance degradation is minimized while high energy savings is achieved. We also address interactive workloads, which account for the bulk of the processing demand on modern mobile or desktop systems. We propose Interaction-Aware Dynamic Voltage Scaling (IADVS) for CPU and Interaction-Aware Memory Energy Management (IAMEM) for memory. The IA framework relies on automatic correlation of user-initiated tasks with the demand placed on CPU and memory to accurately predict power states for CPU and memory. Both mechanisms achieve maximal energy savings while minimizing the impact on the application’s performance.
CHAPTER 1

INTRODUCTION

Computer system designers face two challenges: energy and performance. The spectrum of system design considerations ranges from ultra high performance on one side, in which energy consumption is not the first consideration, all the way to ultra-low power design, where performance expectations are very limited. The middle of the range is occupied by systems where both power and performance play a role in system design. Portable system designers are faced with a user demand for performance, functionality, and better user interfaces along with a longer battery life. Designers of servers and desktop systems previously focused almost entirely on performance, since energy usually is not a constrained resource. However, this trend has been changing recently since researchers have realized the positive financial and environmental implications of energy conservation for stand-alone servers and server clusters [2, 6, 12, 35]. The challenge of designing energy efficient systems lies in understanding the role of user interaction and application behavior in energy consumption and in providing an energy-performance schedule that adequately accommodates demand of the user and application.

The main goal of energy management is not degrading performance while saving energy. Since each execution environment may have a different perception of performance, each needs to be carefully studied in order to maximize energy efficiency while maintaining that particular environment’s performance expectations. For large computing infrastructures, the resulting designs will lower energy consumption and consequently reduce operating costs through reducing the energy bills and cooling costs. The design will also benefit portable and ultra-portable systems by an increase in battery life.

Existing work that addresses performance and energy optimizations ranges from hardware optimizations to application transformations. Figure 1.1 shows a typi-
Optimizations at lower levels are usually complementary to higher-level optimizations. The lower levels also provide the interfaces for managing both energy and performance at higher levels. Observing that the resulting behavior of all system layers is usually a response to user actions and/or application behavior, we deploy our research in two directions: 1) optimizing energy efficiency for non-interactive systems by utilizing the context in-between application and operating system, and 2) optimizing energy efficiency for interactive systems by utilizing the correlation between user interactions and application demand.

1.1 Energy Management for Server Systems

Modern computer systems contain many energy-hungry components. However, due to the recent advances in capacity and bus frequency, the energy consumption of main memory is now surpassing other components. Bus frequencies are steadily rising to provide higher data bandwidth to the data-hungry multi-core processors. In addition, the demand for memory capacity is growing at an even faster rate to
accommodate the data-centric applications that dominate in today’s data centers. For example, the recent EMC Symmetrix DMX3000 storage system can be configured to have up to 256 GB of main memory [13] and the IBM eSeries p5 595 server is configured with 2TB of main memory [27]. Consequently, a significant portion of total energy is consumed by main memory. For example, from the power measurement shown in Table 1.1, as much as 40% of the system energy is consumed by the memory subsystem of a mid-range IBM eServer machine [26].

<table>
<thead>
<tr>
<th>IBM p670</th>
<th>Power (Watt)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processors</td>
</tr>
<tr>
<td>4 cores, 16GB</td>
<td>384</td>
</tr>
<tr>
<td>8 cores, 128GB</td>
<td>840</td>
</tr>
</tbody>
</table>

Table 1.1: Power consumption breakdown for IBM p670 eServers with different configurations.

Energy optimization of the memory subsystem is addressed at both hardware and software levels. At hardware level, energy efficiency is primarily gained through advances in manufacturing processes, which produce more dense modules and lower per-bit energy consumption. Hardware designers have also introduced low-power modes to modern SDRAMs and exposed them to system software, enabling OS-driven energy management. The operating system has a detailed view of the running applications and the demand they place on the system, and therefore allows more sophisticated energy management mechanisms to be implemented. While the additional context available at OS level provides better energy management possibilities, the task of designing an efficient energy management technique is not easy due to the long power-state transition delays that could be exposed to the application execution.

In modern operating systems, main memory is dynamically distributed between buffer cache and virtual memory available to the running applications. In server systems, most memory space is used for caching the previously accessed file pages from the client access request. Buffer cache thus dominates in server systems and it can occupy as much as 77% of the total available memory on desktop computers and
much more on storage servers [24]. Chapter 3 shows that in a server system where the majority of file-I/O accesses to buffer cache are made through system calls, the operating system knows when the access completes and can put the memory into a power saving state immediately. While this simple method saves energy, the transition delays upon the next access can significantly degrade memory performance. Subsequently, we focus on eliminating the delays incurred by power state transitions while maintaining high energy savings for buffer cache. Subsequently, we propose several energy management mechanisms for buffer cache in Chapter 3 and make the following contributions:

1. Quantifies the need for sophisticated techniques for reducing overheads of memory power state transitions in buffer cache.

2. Designs accurate mechanisms for hiding delays in buffer cache energy management for all access types.

3. Quantifies the benefits and overheads of proposed mechanisms and heuristics through detailed simulation.

1.2 Energy Management for Interactive Systems

In interactive systems, one challenge of efficient energy management lies in increasing the energy efficiency of the entire system rather than just an individual component. Introducing any execution delays through component-wise energy management is usually detrimental to the entire system. If one component runs slowly, the entire system may stay on longer, nullifying the energy saved from the specific component, or even increasing the energy consumption of the entire system. However, executing all tasks at the highest performance level is not necessary and may not reduce system delays, as is the case with interactive and real-time tasks. Therefore, it is critical to distinguish between tasks that do not prolong program execution and those tasks that impose delays on the whole system when executed at lower performance levels. The former tasks can be executed at lower performance settings to save energy,
while the latter must be executed at the highest performance setting to minimize the system-wide performance degradation and energy consumption.

Another challenge in designing efficient energy management is providing the energy optimization transparently. In mobile and desktop systems, a majority of the demand placed on the system is in direct response to user input. To accurately correlate user interactions with performance demand, it is necessary to obtain a fine-grained interaction and execution context in complex GUI environments without requiring any user involvement or application modifications [7, 8].

Performance and the overall energy efficiency may suffer if the overheads of energy management are not addressed properly, since the entire system has to stay on longer and consume additional energy. In addition, the user may even be irritated to the extent that he or she completely disables the energy management mechanisms. Fortunately, maximal performance is usually not necessary to meet the user’s performance expectations. For example, CPU or I/O bound tasks in interactive applications may not be noticeably degraded when the memory is operating in a low-power state. Furthermore, the perceived performance of real-time applications such as video players, games, or teleconferencing may not be affected when varying a given device’s power state, as long as the system maintains perceptual continuity for the user. Therefore, the key to transparently providing energy efficiency lies in 1) accurate prediction of upcoming task demand and 2) providing the minimum required performance to meet the upcoming demand without introducing observable delays.

1.2.1 Processors

Today’s CPUs are manufactured with support for energy management through Dynamic Voltage Scaling (DVS). DVS allows software to dynamically reduce CPU voltage, which in turn reduces the CPU’s energy consumption, since it is proportional to the square of the voltage, i.e. $E \propto V^2$. However, as voltage is decreased, the maximum operating frequency is also reduced, resulting in a reduction in performance. In Chapter 4, we propose Interaction-Aware Dynamic Voltage Scaling
(IADVS), a mechanism for matching CPU frequency to task demands and users’ performance expectations. Compared to the existing coarse-grained approaches to interaction capture [28], IADVS’s fine-grained interaction capture yields more accurate predictions of upcoming performance levels demanded by user-initiated tasks. The high prediction accuracy ultimately results in more energy-efficient executions of the upcoming tasks. Subsequently, Chapter 4 makes the following contributions:

1. Identifies and quantifies the need for fine-grained task classification.

2. Integrates mouse and keyboard interaction capture mechanisms;

3. Proposes IADVS and performs a detailed evaluation of IADVS against existing state-of-the-art DVS mechanisms through simulation.

4. Implements IADVS and evaluates its impact on the energy consumed by the CPU, as well as the entire system, on real hardware.

1.2.2 Memory

The demand for higher memory capacity is not limited to the servers in data centers. Even portable computers are experiencing rapid growth in memory capacity to accommodate user demand for richer multimedia experiences and support for processing capability enabled by high performance processors and video cards. As a result, current portable systems are now commonly sold with 4GB of main memory or more. The memory capacities previously in the realm of server systems are now commonplace in personal computing. In Chapter 5, we propose Interaction-Aware Memory Energy Management (IAMEM), a highly accurate and transparent mechanism for memory energy management in interactive systems. Compared to the existing mechanisms: (1) IAMEM provides energy optimizations within the running process that the user is interacting with as well as all other processes in the system, as compared to previous approaches that only improved energy efficiency of memory occupied by processes waiting for execution [18, 24]; (2) IAMEM is a unified approach that addresses energy efficiency of both buffer cache and virtual memory,
while previous approaches only proposed individual solutions for either buffer cache or virtual memory [18, 24]. Subsequently, we make the following contributions in this chapter:

1. Identifies and quantifies the memory behavior of common interactive applications and show large opportunity for improvement.

2. Utilizes high-resolution context of user interactions to accurately predict memory demand for tasks initiated by the user.

3. Proposes IAMEM, a unified energy management mechanism for the entire memory subsystem.

4. Compares IAMEM with the existing state-of-the-art mechanisms through a detailed study.
CHAPTER 2

BACKGROUND

2.1 Energy and Power

Energy is commonly referred as the ability that a physical system has to do work on other physical systems and is the total amount of work a system performs over a period of time. Power is the rate at which work is performed or energy is converted. In the context of computer systems, these concepts are viewed with respect to the activities associated with running programs. Power is the rate at which the computer consumes (or dissipates in the form of heat) electrical energy while performing these activities, and energy is the total electrical energy the computer consumes (or dissipates as heat) over time [39].

It is important to distinguish the terms power and energy since reducing power dissipation does not necessarily translate into energy savings. A program may take longer to complete at the lower performance due to power reduction, and thus the overall energy consumption may not be reduced proportionally or even increased as the power is reduced. On the other hand, applying the highest power and performance can shorten execution time and thus the system can be put into idle power-saving state sooner, which sometimes may reduce overall energy consumption.

There are various scenarios that demand power reduction or energy reduction or both at the same time. For example, in mobile and portable systems where battery lifetime is constraint, conserving energy is of higher priority since it can prolong battery life. However, in large-scaled server systems, management of both power and energy consumption is crucial. First, to increases system reliability while cutting cooling and maintenance cost, power management is critical for controlling thermal dissipation. Secondly, it is also important to reduce energy consumption
due to the demand for electric budget and eco-friendliness.

2.1.1 Dynamic Power and Static Power

The power consumption of a Integrated-Circuit component consists of two components: dynamic power consumption and static power consumption. Dynamic power consumption results from the active circuit activity such as an ALU operation in the CPU or a read access to the memory [39]. The following equation illustrates the CMOS circuit dynamic power consumption:

\[ P_{\text{dynamic}} \propto \alpha CV^2F \]

\( \alpha \) is the activity factor, i.e., the fraction of the circuit that is switching, \( C \) is the switched capacitance, \( V \) is the supply voltage, and \( F \) is the clock frequency. Subsequently, reduction of dynamic power consumption can be achieved by reducing either of the four components in the equation. Dynamic frequency scaling (DFS) is such a technique that is able to lowering operating frequency at program runtime. However, since performance is degraded, DFS does not always reduce energy consumption as discussed before. Therefore, another technique called dynamic voltage scaling (DVS) is more widely applied, that lowers the frequency and voltage simultaneously. Since dynamic power is quadratic in voltage and linear in frequency, a small drop in voltage and frequency can have a substantial reduction to power and energy consumption.

A computer device consumes static power in addition to dynamic power, which is also known as idle power or leakage power. Leakage power exists no matter the device is actively executing program or being idle, and accounts for all power in the device other than dynamic power [39]. The leakage power consumption is calculated as the product of supply voltage (\( V \)) and the leakage current (\( I \)) in the following equation:

\[ P_{\text{leakage}} = VI \]
### 2.2 Processor Power States

CPU is the central component in a computer system which plays the decisive role in determining the performance and power efficiency of the overall system. To improve its energy efficiency, hardware designers have equipped CPUs with multiple power states for both active state and idle state.

While actively executing instructions, a CPU is in operating mode (C0 state). The frequency and supply voltage of each core in the CPU can be set to appropriate levels by DVS techniques to save dynamic power consumption. Each pair of frequency and voltage setting is defined as a performance-state (P-state). Hardware and software can be configured to dynamically change a given core’s P-state according to the current workload intensity. Table 2.1 shows the P-states that the a quad-core AMD Phenom II X4 940 CPU supports [1]. The Phenom II CPU comes with the Cool’n’Quiet (C’n’Q) 3.0 power saving technology, that provides four P-states for each CPU core. The frequency-voltage transition for one core is performed through a low-latency P-state transition, on the order of 10 µs.

When not executing code, the CPU will enter one of the low-power idle states
Table 2.2: The C-states supported by a AMD Phenom II X4 940 CPU with associated transition latencies and the state descriptions.

called C-states. The introduction of C-state is used to reduce leakage power consumption by stopping the clock signal and interrupts of idle units inside the CPU. The higher C-state the CPU enters, the more units are stopped and more voltage is reduced, and thus the more energy is conserved, at the cost of the more time to wake up the CPU however. Table 2.2 shows the C-states and state specifications that a AMD Phenom II X4 940 CPU supports.

2.3 CPU Power Management

Dynamic voltage scaling (DVS) targets at manipulation of a processors frequency and voltage as programs are executing to save dynamic power and energy. According to Section 2.1, a processor’s dynamic energy consumption can be calculated as the following equation, where T represents the running time and W is the workload’s CPU cycles which is assumed to be a constant.

\[ E \propto \alpha CV^2 FT \propto \alpha CV^2W \]

Since the energy is only proportional to the square of voltage, scaling down the CPU’s frequency and voltage are generally more energy efficient than running it at full speed and then idling for the remaining duration (which is sometimes referred as race-to-idle). A CPU with relatively light workloads can therefore be slowed down to certain extent with no significant performance penalty.

If a workload’s CPU requirement and processing deadline were known prior to its execution, the CPU can be simply run at a speed setting just fast enough to finish the workload by its deadline. For example, if a task requires 10 million CPU
cycles to execute with a deadline of 100 ms, the clock frequency and supply voltage can be theoretically set to 100 MHz. Unfortunately, most modern systems lack such information from existing applications, and developers for new application would be unwilling to or unable to expose accurate information to the system. Therefore, the key of a DVS mechanism is how to predict workload information with acceptable accuracy. Based on the workload granularity, system-level DVS techniques can be generally classified into interval-based approaches and task-based approaches.

2.3.1 Interval-Based Approaches

Interval-based DVS mechanisms divide a workload into fixed time intervals, and predict the CPU utilization for the next time interval based on the historical intervals. One of the earliest approach, PAST, was proposed by Weiser et al. [40], which predicts the upcoming CPU utilization to be the same as on the most recent interval’s. If the predicted utilization is above a certain threshold, it speeds up the CPU; and if the utilization is below another threshold, it slows down the CPU. Its major drawback is the inferior accuracy and possible thrashing since its prediction is based solely on the last interval. Govil et al. [16] extended PAST by considering a larger number of recent intervals and proposed several prediction algorithms to improve the accuracy. For example, their LongShort method averages the 12 most recent intervals utilizations, while another method PEAK monitors the recurring patterns for CPU utilizations, with special attention to short bursts of high utilization, and sets the CPU speed accordingly.

2.3.2 Task-Based Approaches

Interval-based approaches are effective for regular workloads due to its simplicity but may suffer when workload patterns become more complex and irregular. Task-based DVS mechanisms, on the other hand, characterize the complete CPU workload rather than the periodic CPU utilizations, and thus improve their accuracy. Task-based approaches can be further classified into two subcategories, the intertask and
intratask variants. The intertask variants assign a fixed speed for the entire task execution. Weissel et al. [41] proposed a mechanism that decide the CPU frequency based on the hardware events. Their approach utilizes the performance counters to monitor the hardware events associated with performance and energy consumption, and attributes them to separate processes. Whenever a process is scheduled to run during context switch, it adjusts the CPU frequency to the minimal one that can provide the user-defined performance threshold based on previously measured events. To further improve the workload estimation, Flautner et al. [15] classified workloads as interactive tasks and producer-consumer tasks, each of which is applied with a separate DVS algorithm. Based on the workload prediction, interactive tasks can be slowed down up to the user perception threshold (between 50 ms and 100 ms) while producer tasks can be stretched to the beginning of their associated consumer tasks. To recover from prediction error, their algorithm switches the CPU to the maximal speed as soon as a panic threshold is reached.

Intratask approaches adjust the CPU frequency and voltage within each task. Lee et al. [25] proposed an prototype approach called run-time voltage hopping that splits each task into fixed-length timeslots and assigns each timeslot the lowest speed allowing it to complete within its preferred execution time. A task’s preferred execution time is measured as the task’s worst case execution time minus the elapsed execution time up to the current timeslot. PACE [29] and Stochastic DVS [17] improved this approach by choosing a speed for every cycle of task execution based on the probability distribution of the task workload measured over previous cycles. Both approaches contain simplifying assumptions that the energy consumption is proportional to the square of either voltage [17] or frequency [29], which are not guaranteed to hold in real systems. Also, voltage scaling are not free and the transition between two performance states require both time and energy overheads [3]. Therefore, over-frequent switching can often carry significant transition overheads and may not be practical. Careful considerations have to be taken to minimize the overheads due to voltage and frequency scaling.

Similar to [15], Lorch et al. [28] also conducted an extensive study on month-long
traces of user interface applications to evaluate various DVS algorithms including PACE. Compared to [15], they proposed a low-overhead heuristic for accurately determining the completion of a user interface task. Through comprehensive evaluation, they suggested that compared with no separation of user interface tasks or separating tasks only by interaction types (mouse or keyboard events), separating tasks by additional information such as interaction categories and applications, can provide better prediction accuracy of task length, while only gaining marginal energy savings.

2.4 Memory Power States

Modern computers are relying on large-capacity and high-speed main memory to provide desired performance for data intensive applications. SDRAM is widely used for main memory in the form of Double-Data-Rate (DDR), followed by DDR2 and DDR3, which are the mainstream DRAM architecture in current computer systems. DDR series memory is packaged into DRAM modules, each of which commonly consists of two ranks. Each rank includes a number of physical devices and, in the case of server systems where timing and data integrity are a critical consideration, the registers and phase-lock loop (PLL) devices.

The smallest unit of power management in DDR family is the rank and all devices in a rank are operating at the same power state [30]. For simplicity, we only consider four power states that a rank can operate in: (1) I/O state: the state when memory is reading or writing data; (2) Precharge state (PRE): the active idle state where the next I/O can take place immediately at the next clock cycle; (3) Powerdown state (PD): several subcomponents of a rank are disabled to save power in this state, such as I/O buffers, sense amplifier, row/column decoder, etc; (4) Self Refresh state (SR): in addition to the Powerdown state, the external clock, on-die termination, as well as the PLL and registers are disabled in this state to reduce power consumption even further.

Despite of the low-power states (PD and SR) provided, memory I/Os can only
Table 2.3: Power specifications for a rank consisting of 8 Micron 1Gbit DDR2-800 devices.

be performed with memory in the high-power state (PRE); therefore, the rank in
a low-power state has to be transitioned to PRE state before performing any I/O,
potentially exposing a high resynchronization delay to the application. Figure 2.1
illustrates the latencies associated with the power state transitions for a registered
Micron DDR2-800 rank which is typically used in server systems. Table 2.3 also
presents its corresponding power states [31, 32] with the total rank power being
calculated as the product of the per-device power and the number of devices in the
rank, plus the power consumed by the registers and PLL.

As a counterpart to server memory, Figure 2.2 presents the power specifications
for an unregistered Micron DDR3-1066 rank normally used in desktop and portable
systems, including the power consumptions, the power state transitions, and the
associated resynchronization latencies [32, 31]. It should be noted that there is no
register and PLL power for each power state in this case.
2.5 Memory Power Management

As applications become more data-centric, memory requirements for both speed and capacity have been increasing continuously. Subsequently, lots of research work has been done about memory power management. Power management techniques for main memory can be grouped into hardware approaches, software approaches, and the combination of both.

2.5.1 Hardware Approaches

Hardware level approaches generally utilize the memory controller to monitor the memory traffic as well as memory access patterns, and make power state transitions for specific memory devices based on the observed energy-saving opportunities. Lebeck et al. [23] studied the interaction of page placement with static and dynamic hardware policies to reduce memory power dissipation. The cooperation between the hardware and the OS was also studied in [23]. Pisharah et al. [36] proposed an approach to save memory energy by introducing a hardware unit called Energy-Saver Buffers to hide the resynchronization costs when reactivating memory modules. Fan et al. [14] further investigated memory controller policies for manipulating DRAM power states in cache-based systems and developed an analytic model that approximates the idle time of DRAM chips using exponential distribution. Similar to DVFS techniques applied to the CPU, Deng et al. [11] proposed MemScale, a hardware scheme to increase memory energy efficiency by applying DVFS to the memory con-
controller and applying DFS to the memory channels and DRAM devices. They also proposed several OS policies to guide MemScale that determines the DVFS/DFS mode of the memory based on the current need for bandwidth, energy savings, and the performance degradation that applications are willing to withstand.

Besides viewing memory from the CPU side, Pandy et al. [34] showed that significant energy is consumed when memory is actively idle during DMA transfers and proposed two performance-directed techniques that maximize the utilization of memory devices by increasing the level of concurrency between multiple DMA transfers from different I/O buses to the same memory device. Based on this work, Yue et al. [42] targeted the buffer cache in data servers, and evaluated the energy efficiency for various buffer cache replacement algorithms under real-world parallel I/O workloads through simulation.

2.5.2 Software Approaches

Hardware-level power management may suffer from inaccuracy and may cause unexpected performance degradation. Software-level power management, on the other hand, can provide more detailed context of execution to make timely power state transitions. Delaluz et al. [10] proposed a compiler-directed approach to cluster the data across memory banks, detect memory module idleness and insert power-state transition instructions into a program by offline profiling. They also proposed several hardware-assisted approaches to predict module inter-access time. However, compiler-directed schemes can only work on a single application at a time and demand sophisticated program analysis support. Delaluz et al. [9] also proposed an operating system based solution where the OS scheduler directs the power mode transitions by keeping track of module accesses for each process in the system. This approach was implemented in a full-fledged OS to show energy efficiency at no extra hardware in a multi-programmed real platform.

Subsequently, Huang et al. [18] proposed Power-Aware Virtual Memory that manages the power states of memory devices on per-process basis. PAVM records the memory footprint in terms of virtual memory pages for each process and makes
power-state transitions upon each context switch time. Based on PAVM, a SW-HW cooperated mechanism [20] was proposed to combines PAVM and the underlying hardware. In the work, they employed a hardware power management unit (PMU) in memory controller to monitor the memory traffic and predict the future access pattern. In addition, PAVM was also used in [20] to provide the PMU with the current system state in order to make more accurate power managing decisions. Huang et al. [19] proposed memory reshaping mechanisms that coalesce short idle periods into longer ones through page migration which can be exploited by existing techniques to save extra energy by transitioning memory into deeper power-saving state. To maintain memory performance, Li et al. [27] proposed a technique that provides a performance guarantee for existing power management algorithms by temporarily stopping power management and running memory at full performance. They also proposed a new control algorithm that dynamically adjusts its power management thresholds periodically, based on available slack and recent workload characteristics.

Memory energy management at software level requires support from hardware to control memory power states and monitor system behavior in detail. Figure 2.3
shows Intel’s most recent Core i CPU with integrated memory controller for low-overhead power state management and monitoring. The Core i CPU provides the dedicate registers to put DRAM ranks into Self Refresh state by enabling "Clock (CKE) Low" signal. In addition, the Core i CPU provides a mechanism called Dynamic Memory Rank Power Down which is able to put memory ranks into Powerdown state automatically after a specified memory idle time has elapsed [22]. Subsequently, system software only needs to set up the associated register with the desired timeout value to enable this feature. The integrated memory controller will then perform the power state transitions automatically, after the preset timeout expires.

Detailed system monitoring is further provided by the Core i CPU through a set of registers called performance counters, which are crucial for monitoring memory accesses. Under normal operations, virtual memory accesses are invisible to system software, while only occasional page faults results in OS being invoked. Performance counters, however, enable the software to monitor memory activity when applications are performing memory I/Os, such as the number of CPU cache misses that result in main memory accesses. However, exact timing of each memory request, that would allow us to determine memory access burstiness, is not available.

2.6 Workloads Characteristics

The most important issue that needs to be addressed for system-level energy management is how to obtain the workload’s CPU and memory activity with reasonable timeliness and accuracy as well as low overhead. The challenge mainly resides in the unpredictability of future execution times when workloads become irregular even with the knowledge of past execution times. Microarchitectural features such as pipelining, out-of-order execution and multi-threading make it difficult to predict execution times in real systems statically. As the current trend of computer system design involve more user interaction with the application, execution environment with or without user interaction become a major factor in determining the work-
load regularity, and workloads can thus be classified into interactive workloads and non-interactive workloads.

2.6.1 Non-Interactive Applications

Non-interactive applications usually exhibit regular workloads such predictable CPU utilization and repeatable memory access patterns. These regularities can be captured, correlated, and exploited in the decision-making process of energy management mechanisms. Furthermore, it takes light effort to analyze energy-performance tradeoffs in non-interactive applications since the degradation in performance due to energy management is usually directly translated into delay in the applications execution time.

Operating system can provide useful hints for characterizing workloads. Physical memory is allocated between virtual memory (VM) and buffer cache by modern OS. The buffer cache improves the file system performance by caching the previously accessed disk blocks. As data intensive applications become prevalent, the demand placed on the file system increases. As a result, buffer cache in modern data servers can span terabytes of physical memory [27]. Figure 2.4 shows a profile of memory usage during the execution of several I/O bound applications in a Linux-x64 system with 8 GB RAM. As much as 92% of memory space is allocated for buffer cache while less than 27% is used for virtual memory and other system memory. It is evident that energy management for buffer cache is crucial to improving the overall memory energy efficiency. In Chapter 3, we explore server applications and propose mechanisms that utilize OS file system to provide necessary context of memory access pattern to buffer cache and thus to make timely power management decisions. Memory power state transitions are issued at certain locations inside I/O handling routines to hide transition latency behind kernel processing so that performance degradation is minimized with maximal energy savings.
Figure 2.4: Memory profile for I/O bound workloads running in a system with 8 GB RAM.

2.6.2 Interactive Applications

The workloads and timings in interactive applications are less predictable as they depend on the user interacting with the application in addition to the application itself. Therefore the design of energy management for interactive applications is more challenging. Slowing down performance-oriented tasks prolongs execution, exposes delays to the user, and potentially increases energy consumption since the entire system spends more time processing a given task [38].

Fortunately, a user spends a majority of time performing low-demand tasks, such as typing and reading, that do not require high performance. User’s response time presented in interactive applications may turn out to be a benefit, since it can potentially hide execution delays introduced by energy management mechanisms. The user’s response time is dictated by the perception threshold whose average value has been shown to range between 50 ms and 100ms [37]. Any task having its completion earlier than the perception threshold are likely to be imperceptible to the user, while the system should idly wait for the user to initiate the next task. Tasks that complete before the perception threshold is reached do not impact the speed with which the user interacts with an application, and thus do not prolong the application’s execution time. Figure 2.5 shows the cumulative distribution of the execution
times of interactive tasks in Linux GNOME desktop environment. It is observed that over 92% of the total tasks can be completed within 100 ms (the upper limit of the perception threshold) when the system runs at the maximum performance. All of those tasks can be potentially executed at a lower performance, improving the system’s energy efficiency. The remaining 8% of tasks are relatively long tasks which cannot fit within the user’s perception threshold. To prevent performance degradation and potential increase in overall system energy consumption, those tasks should be executed at the maximum performance.

Subsequently in Chapter 4 and Chapter 5, we propose several Interaction-Aware energy management mechanisms (IADVS for CPU and IAMEM for memory), that treat the perception threshold as the deadline for processing interactive tasks: Tasks with the execution time below the perception threshold can be executed at a lower performance level, extending their execution time up to, but not beyond, the perception threshold. The lower performance level improves energy efficiency while meeting the user’s expectations of interactive performance. By preserving the perception deadline, we prevent the users from noticing any impact on the system from energy management mechanisms. Subsequently, user behavior is unaltered and we can focus on performance and energy metrics without the need for controlled user studies.
CHAPTER 3

DELAY-HIDING ENERGY MANAGEMENT MECHANISMS FOR DRAM

Hardware-level power management may benefit from fine-grained, low-level information provided by the memory controller that monitors the memory activities on each device and turns the devices off based on the detection of idle periods. However, hardware is generally unaware of higher-level information, such as which file is accessed by which process or the layout of file blocks in buffer cache. As a result, hardware-level power management may suffer from inaccuracy and can expose the application to unexpected performance degradation. System-level power management, on the other hand, can provide the necessary context to make timely power-state transitions and reduce or eliminate performance losses.

The simple approach to preventing delays is to keep all memory currently owned by a given process in a high-power state. In the case of virtual memory [18], only memory devices used by the newly-scheduled process are powered up during the context switch and remain on during the execution of the given process to prevent any performance degradation. Per-process power management has been further extended to buffer cache [24], where memory devices that contain the buffer pages used by the running process is powered on and kept on during the process execution. While per-process approaches provide significant benefits in multitasking environment, they do not address memory management issues of server environments, where usually only one server process is running. In this scenario, all memory devices that belong to this single process would always remain on, eliminating any possible energy savings.

Unlike virtual memory, every access to buffer cache through system calls is visible to the OS, allowing for more sophisticated mechanisms that can take advantage of this additional context. By separating the VM power management from the buffer cache power management, we can achieve better energy efficiency and performance
for buffer cache through leveraging I/O system calls. In this study, we only consider standard buffer cache accesses made through systems calls; any file that is memory mapped to the VM space is managed by the VM power management, and is not impacted by the buffer cache power management. Furthermore, separation of the buffer cache and virtual memory into separate spaces in physical memory can also improve overall energy efficiency [24]. The mechanisms proposed in this study likewise allocate the buffer cache in a separate portion of physical memory to prevent interference from VM accesses.

3.1 Design

3.1.1 Exploiting File-I/O System Calls

A file-I/O system call goes through several steps in the kernel before the actual buffer cache data is read from physical memory, as shown in Figure 3.1. Upon the kernel entry, the file table is checked to get the file identifier (inode), and the inode is used to access the virtual file system (VFS). Each requested block of the file I/O is looked up in the hash table that tracks all blocks in the buffer cache. If the block is found in the cache, the kernel invokes _copy_to_user routine to copy the block from the kernel space to the destination location in user space (_copy_from_user is called for write operations), and the corresponding I/O request is sent to the memory controller.

Energy consumption can be reduced by putting the rank into a low-power state during the idle period between two consecutive I/Os. To avoid powering down rank that will be immediately accessed, power management techniques for VM use a timeout scheme that usually waits for a few nanoseconds for more memory I/Os to arrive before powering down the rank [14]. The access granularity for file I/O is coarser: the idle period between two consecutive memory I/Os in buffer cache is on the order of hundreds of microseconds. Therefore to increase energy savings further, the mechanisms proposed in this study immediately power down the accessed ranks as soon as a file-I/O finishes.
Due to the high resynchronization cost, it is critical to determine when to turn on a rank upon the access request to the rank. The best scenario occurs when the transition finishes just before the request arrives at the memory controller. In this situation, no energy is unnecessarily consumed while waiting for the request and the transition latency is completely hidden. Late transitions delay memory I/Os and transition latencies are partially or fully exposed. The overall system energy consumption can also increase due to the longer service time. Finally, if the rank is turned on earlier than request arrival, the delay is avoided at the cost of excess energy consumed when waiting for the request. As shown in Figure 3.1, there are several stages where the rank can be turned on. The goal is to initiate and complete the transition before a memory request arrives at the memory controller.

We have instrumented the Linux 2.6.20 kernel and measured the time spent within each I/O call subroutine on a 3.0GHz AMD processor. We first investigated the time between block-locating routine and _copy_to_user routine, where 90% of the system calls complete in fewer than 120 ns, far from hiding the 500 ns transition.
latency from Self Refresh state. Transitions initiated at this time will be too late, exposing almost 76% of transition delays. We then investigated the time between file-id retrieving routine and _copy_to_user routine, finding that for 90% of I/Os the time is longer than 500 ns, and therefore, can fully cover the transition latency. Furthermore, 85% of I/Os encounter times between 500 ns and 550 ns. These key observations strongly suggest that turning on ranks right after file-id retrieving routine can preserve 90% of performance with hardly any energy penalty. Since file-id retrieving routine only takes a few instructions in the beginning of a file-I/O system call, we will refer to this point as ”system call entry” for the remainder of the discussion.

3.1.2 All-Ranks-On for Delay Reduction

Physical memory mapping is not available at the system call entry, and performing a fully fledged translation is not feasible at this point. Subsequently, we propose a simple ALL-Ranks-ON mechanism (ALL) that will naively turn on all ranks containing buffer cache data at the system call entry. The ALL mechanism provides high performance for memory accesses since all the ranks are ready before a memory request arrives and can serve the request without delays. However, file blocks requested by a I/O generally reside on a single rank. Other ranks that do not hold the data are unnecessarily turned on and consume energy during the I/O operation. While energy savings are obtained by keeping memory in the low-power state during the idle periods between I/Os, these periods can be small in data intensive applications. Therefore, the ALL mechanism may offer less than desired energy savings. Nevertheless, the ALL mechanism represents a lower bound on the exposed transition delay for all the proposed mechanisms that power on the rank at the system call entry.
3.1.3 Predicting Individual Ranks

To reduce the energy consumption of the ALL mechanism, we will propose several mechanisms that predict the most likely rank to be accessed and only turn on that rank at the system call entry. The simplest mechanism is the Most-Recently-Accessed (MRA) mechanism that predicts the current I/O will access the same rank as the last I/O. With sequential cache allocation, buffers are allocated in the order they are accessed, and an entire rank is filled before moving on to the next one. In case of sequential access applications that generally read an entire file sequentially, the MRA mechanisms will achieve high accuracy since it is highly probable that the current referenced blocks are allocated to the same rank as the previous one.

The MRA mechanism can potentially save more energy since only one single rank is predicted and turned on for each I/O. However, in random access applications, its accuracy could be drastically reduced, which increase both delays and energy consumption. When an incorrect rank is turned on, the transition of the actual accessed rank is delayed until the request arrives at the memory controller, exposing the full transition delay. To refine the prediction granularity, and hopefully increase the accuracy, we can distribute the last rank information to each individual file. Subsequently, we propose the per-File MRA (F-MRA) mechanism that predicts and turns on the last used rank by each accessed file. To implement F-MRA, we slightly modify the file table in the kernel to include the information of last used rank for each file (shown in Figure 3.2a.)

The F-MRA mechanism will benefit applications that access many files concurrently but accesses within each file are sequential or at least grouped in the same rank. However, problems may arise when the accessed file is spread among several ranks due to replacements or access patterns that did not result in contiguous placement in the rank. Therefore, we consider probabilistic approaches and propose the per-File-Most-Frequently-Accessed (F-MFA) predictor that keeps track of the per-file access frequencies. To count the rank access frequency for each file, we add an array of counters in the associated file table entry (shown in Figure 3.2b.). Each of
the counters corresponds to one rank and records the number of completed accesses in that rank. The counter value is incremented upon an access to the corresponding rank. Each time a file is accessed, the F-MFA mechanism predicts that the file’s most frequently used rank (with the largest counter value) will be accessed.

Frequency based approaches suffer sometimes from hot blocks, where a block is initially accessed very frequently and not accessed in the future. As a result, we propose another mechanism based on file distribution among ranks. The per-File-Data-Rank-Density (F-DRD) predictor keeps track of the number of blocks that each rank contains for a given file. The data structures in each file table entry are identical to the F-MFA mechanism, as shown in Figure 3.2b. The only difference in the implementation is that the counters are updated upon each buffer placement and replacement in the given rank for F-DRD. Every time a new block is allocated to a rank, the corresponding counter is incremented by 1. When a block is replaced out of cache, the counter corresponding to its original rank is decremented by 1. Similarly to the F-MFA mechanism, the F-DRD mechanism retrieves the file distribution from the file table and turns on the rank that contains the most blocks of the currently accessed file.

3.1.4 Increasing Prediction Coverage

Both F-MFA and F-DRD mechanisms predict only one rank upon a system call. As a result, they can miss turning on the accessed ranks and expose the delays. To reduce the misprediction, we increase the number of ranks that each mechanism
turns on. The simplest extension to those mechanisms is setting a desired threshold \( T \). For F-MFA, \( T \) represents the minimum access frequency that all powered-on (active) ranks should provide. For F-DRD, \( T \) represents a minimum file fraction to be present in active ranks. For each I/O, F-MFA will select to turn on the smallest number of ranks that can provide the total access frequency greater than \( T \). For example, if we set the threshold \( T \) to 90% and the file is distributed among 3 ranks with access frequencies of 62%, 30% and 8%, F-MFA will turn on the first two ranks, providing the minimum desired frequency of 90%. Similarly in case of F-DRD, it will turn on the minimum number of ranks that provide the total file fraction greater than \( T \). For example, if we want 90% of the file to be present in active ranks and the file is distributed among 3 ranks with the block distribution of 62%, 30% and 8%, F-DRD will turn on the first two ranks containing a total 92% of the file blocks.

Setting the threshold to 100% will turn on all ranks that the accessed file resides in, eliminating all delays while keeping ranks that do not contain any portion of the file in low-power states. However, some applications may be able to better tolerate the delay and 100% of accesses to active ranks may not be necessary. For example, the user may desire a global coverage rate of 90% by the predictor, meaning that 90% of the accesses should go to active ranks that are powered on by the predictor. To enable the user desired global coverage rate, we modify F-MFA and F-DRD and propose Adaptive F-MFA (AF-MFA) and Adaptive F-DRD (AF-DRD) that adaptively adjust the threshold for each file, i.e., the minimum required access frequency or block distribution that active ranks should provide. Subsequently as shown in Figure 3.2c, these mechanisms maintain three additional variables per file in the file table: 1) the number of accesses that resulted in hits in active ranks (\( \text{num\_hit} \)), 2) the number of completed accesses to the file (\( \text{num\_access} \)), and 3) the current threshold (\( \text{threshold} \)). Initially, the user presets the desired global coverage rate, and each file’s \( \text{threshold} \) is set to some default value, e.g. 50%. After every access, \( \text{threshold} \) is dynamically adjusted based on the comparison of the local coverage rate (calculated as \( \text{num\_hit} \) divided by \( \text{num\_access} \)) to the global coverage rate. If local coverage rate is lower, \( \text{threshold} \) is increased and otherwise decreased. Overtime,
the local coverage rate will eventually converge to the global coverage rate.

A similar optimization can be implemented for the F-MRA mechanism, resulting in the Adaptive F-MRA (AF-MRA) mechanism. In AF-MRA, we also introduce three additional variables in each file table entry, as well as a stack to maintain the list of MRU ranks. The first two variables maintain the number of accesses and the number of hits as before to calculate the local coverage rate. The third variable \( \text{num}_\text{ranks} \) stores the number of most recently used ranks that the predictor should turn on. Initially \( \text{num}_\text{ranks} \) is set to one, and it is automatically adjusted based on the comparison between local coverage rate and the global coverage rate: if the local coverage rate is lower, \( \text{num}_\text{ranks} \) is increased, and otherwise decreased. Upon each file-IO, AF-MRA will turn on the most recently used ranks indicated by \( \text{num}_\text{ranks} \) and the MRU stack.

### 3.1.5 Improving Energy Efficiency

The proposed mechanisms may turn on ranks that will not be accessed, either by mispredicting a rank to turn on, or by aggressively turning on more than one rank. Subsequently, the wrongly turned-on ranks will be idling in Precharge state during the I/O, and consume excess energy. To address this issue, we propose an early-turnoff optimization that can track the rank to be accessed and turn off the unneeded active ranks at the earliest point to save energy. To efficiently select the rank to be accessed from active ranks, we slightly modify the data structures in the buffer cache hash table. For each table entry, we add an additional variable to record the rank that the corresponding block is placed in. Subsequently, when a block is requested, its containing rank is to be accessed and can be obtained as soon as the buffer cache lookup completes in block-locating routine. Other active ranks will not be accessed and can now be turned off. With this optimization, each proposed mechanism executes the following steps. First, at the system call entry, several ranks are turned on as indicated by the predictor. Second, once the buffer cache lookup is performed, the rank to be accessed is retrieved and all the other ranks that were previously turned on are turned off immediately.
### Table 3.1: Applications and trace statistics.

<table>
<thead>
<tr>
<th>Application</th>
<th>Num. of I/O calls</th>
<th>Num. of blocks</th>
<th>Num. of unique files</th>
<th>Dataset size (MB)</th>
<th>Hit ratio (4GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glimpse</td>
<td>18351930</td>
<td>20574810</td>
<td>319712</td>
<td>4237</td>
<td>84%</td>
</tr>
<tr>
<td>Postmark</td>
<td>17897551</td>
<td>20000000</td>
<td>101421</td>
<td>5047</td>
<td>94%</td>
</tr>
<tr>
<td>TPC-H</td>
<td>98789470</td>
<td>100000000</td>
<td>107</td>
<td>7156</td>
<td>96%</td>
</tr>
<tr>
<td>TPC-C</td>
<td>47461324</td>
<td>95908075</td>
<td>542</td>
<td>4775</td>
<td>97%</td>
</tr>
<tr>
<td>Multi1 (TPC-H, TPC-C)</td>
<td>197410317</td>
<td>200000000</td>
<td>112</td>
<td>7247</td>
<td>96%</td>
</tr>
<tr>
<td>Multi2 (Glimpse, Postmark)</td>
<td>26867682</td>
<td>30000000</td>
<td>366221</td>
<td>8500</td>
<td>86%</td>
</tr>
</tbody>
</table>

3.2 Methodology

We used trace-driven simulation to evaluate our proposed mechanisms and compared them with the existing on-demand mechanism (ISR). Detailed file-I/O traces of each application were collected by a modified `strace` utility that intercepts the system calls from the traced process and records the following information about each I/O operation: access type, time, file identifier, and I/O size. We selected four common server applications: Glimpse, Postmark, TPC-H, and TPC-C. Glimpse is a text indexing and search application. Postmark is a mail server benchmark designed to measure the transaction rates for a workload approximating a large mail server. TPC-H is a decision support benchmark which consists of a suite of business oriented ad-hoc queries and concurrent data modifications. TPC-C is an on-line transaction processing benchmark, simulating a intense computing environment where numerous users execute transactions against a database. We also generated two concurrent traces Multi1 and Multi2. Multi1 is a trace of TPC-H and TPC-C executed concurrently and represents a workload in a database querying and processing system. Multi2 is a trace of Glimpse and Postmark executed concurrently and represents a web server providing mail and web search.

Table 3.1 lists the details of the each trace, such as the number of file-I/Os, the number of referenced blocks, the number of files accessed, dataset size, and the hit rate in a 4 GB buffer cache. Due to the long runtime of Postmark and TPC-H, we selected the first 20 million references for Postmark and the first 100 million references for TPC-H in our traces. The concurrent traces were also truncated.
accordingly; as a result the statistics of concurrent execution is not the sum of the two individual applications.

We augmented a buffer cache simulator [4] with a physical memory simulator to implement the proposed mechanisms. The memory simulator includes a memory controller and 4 DDR2-800 ranks to simulate a total 4 GB memory with the power specification shown in Table 2.3. The entire storage space is used for buffer cache with 4KB block size. Since the traces do not contain the I/O processing time in the kernel, we set the time from the system call entry to _copy_to_user routine to 500 ns, and set the time from block-locating routine to _copy_to_user routine to 120 ns, as discussed in Section 3.1.1. Finally, sequential buffer placement is used to allocate free buffers and the LRU replacement algorithm is used to select the victim block during the buffer replacement.

3.3 Evaluation

3.3.1 Accuracy of Predictors

Figure 4.4 shows the prediction breakdown for the proposed mechanisms. The Hit portion represents the number of transitions of ranks that were correctly turned on by the predictor, while the Miss portion represents the number of transitions of ranks that were accessed but not turned on by the predictor. The sum of hits and
misses is the total required rank transitions in each application, and is normalized to 1. The Wrong portion above 1 represents the number of rank transitions that were initiated by the predictor but not accessed in the subsequent file-I/Os. Therefore, a missed rank transition results in exposed transition delay, while a wrong rank transition consumes excess energy.

Let’s first focus on the predictors (MRA, F-MRA, F-MFA, F-DRD) that only predict to turn on one rank at a file-I/O. Each of these single-rank predictors has the same number of wrong transitions as the number of misses, meaning that a predictor with more hits will have less wrong transitions. For Glimpse, all file blocks are sequentially read and allocated, and as a result, all predictors are almost 100% accurate in predicting ranks with hardly any wrong transitions. For other random access applications, F-MRA achieves the highest accuracy, on average 73% hits with only 27% wrong transitions. MRA also predicts to use last rank, but due to the aliasing among different files, the accuracy drops to 64%. The mediocre accuracy of frequency (F-MFA) and file-distribution (F-DRD) based mechanisms show that temporal information provided in MRA and F-MRA is more important for achieving higher accuracy.

When the files are distributed across multiple ranks, it becomes more difficult to accurately select a single rank. Figure 3.3 shows access frequency to files distributed over multiple ranks. The majority of files in Glimpse are small and, due to sequential
accesses, each of them ends up in a single rank. Therefore, I/O accesses in Glimpse always go to files distributed in one rank, explaining the excellent performance of all the predictors. Files in Postmark are also small, ranging from 50 Bytes to 100 KB, but random accesses prevent contiguous allocation of files to ranks. As a result, files are scattered in multiple ranks and 80% of the I/Os go to the files distributed in two or more ranks. Multi2 is a mix of Glimpse and Postmark and shows the intermediate pattern. In TPC-H, TPC-C and Multi1, large database files are accessed randomly such that the blocks of a large file are allocated across multiple ranks. Therefore, 80%-90% of the I/Os from these three applications go to the files distributed in three or more ranks.

Coverage rate in random-access applications can be increased by turning on more ranks at the system call entry, increasing the likelihood that the file portion accessed by current I/O resides in active ranks. Through experiments of F-MFA’s file-access-frequency thresholds and F-DRD’s file-fraction thresholds valuing between 50% and 100%, we found that the resulting coverage rate is 95% on average for both predictors. Subsequently, we use 95% as the global coverage rate for the study of the adaptive predictors. Figure 4.4 shows that all adaptive predictors (AF-MRA, AF-MFA and AF-DRD) match the preset coverage rate of 95%, significantly improving the coverage of the single-rank predictors. However, higher coverage comes at the cost of more unnecessary transitions, and as a result, wrong transitions in AF-MRA, AF-MFA and AF-DRD increase by 3.2, 1.4, and 1.5 times as compared to F-MRA, F-MFA, and F-DRD, respectively. AF-MRA is the best adaptive predictor due to its smallest wrong transitions. Finally, we set AF-DRD’s global coverage to 100%, and shows the effects of the resulting F-DRD-all mechanism. By turning on all ranks that contain any part of the file, F-DRD-all yields the maximum 100% coverage at the expense of the highest wrong transitions among all predictors.

3.3.2 File-I/O Time

Both performance and energy efficiency of the entire system is affected by performance degradation of main memory. Therefore, the goal of the proposed mechanisms
is to improve the performance while preserving energy savings. The performance of file-I/Os is shown in Figure 3.5. The overall I/O time for each mechanism and application is separated into three components: 1) memory I/O time: time spent on reading/writing data, 2) kernel time: time spent on processing file-I/O system calls inside the kernel, and 3) delay: additional time spent in transitioning ranks from Precharge state to Self Refresh state before any I/O can be performed in physical memory. Since we focus on improving the system call turnaround time, i.e., the time between the system call entry and the return to the application, we exclude the idle time between consecutive file-I/Os when evaluating the performance. Finally, neither the memory access time nor the kernel time are affected, since we do not alter the processing routines.

Ideally, file-I/O time only contains the kernel time and memory I/O time. Therefore, we normalize each mechanism’s time to this ideal time and the delay is shown above the 1.0 line. As we can see, ISR has the most delay since every rank transition is delayed until physical memory access, and as a result incurs 50%-87% performance degradation across all applications. On the other extreme, ALL incurs zero delay by turning on all ranks in the system at the system call entry. The other mechanisms complete the spectrum of tradeoffs between energy and delay. Higher coverage of a predictor (the Hit portion of Figure 4.4) translates into less delay. This is because a rank that was missed turning on will cause an on-demand transition and incur the
full resynchronization latency, while the correct prediction will complete the rank transition before the memory request arrival and fully hide the delay.

For Glimpse, all predictors have almost 100% coverage, thus eliminating all delay. In other random access applications, lower coverage of single-rank predictors results in higher delays. Subsequently, MRA, F-MRA, F-MFA, and F-DRD incurs performance degradation of 19%, 14%, 30%, and 29% respectively. Since all of the adaptive predictors (AF-MRA, AF-MFA, and AF-DRD) achieve 95% coverage rate, they significantly reduce delays, resulting in a mere 3% performance degradation. Finally, F-DRD-all turns on all ranks containing any portion of the file, and therefore preserves the full performance.

3.3.3 File-I/O Energy Consumption

Figure 5.7 shows the energy consumption of physical memory during the processing of file-I/Os from the system call entry until exit. The energy bars are divided into four components: 1) energy consumed for reading/writing data, 2) energy consumed in Precharge state, 3) energy consumed in Self Refresh state, and 4) energy consumed during the transitions between Precharge state and Self Refresh state. We normalize each mechanism’s energy consumption to ISR’s energy (shown as 1.0), since the on-demand ISR is the most energy-oriented power management mechanism when we only consider the energy consumption of main memory. It is worth noting that
delays introduced by ISR may increase the overall energy consumption of the entire system. If we considered overall system energy, the lower delays incurred by the proposed mechanisms would result in better overall energy efficiency.

Energy consumed by memory I/O is the same across all mechanisms for any given application, since the buffer cache reads/writes the same amount of file data. The energy consumed in other states, however, varies among different mechanisms. The largest differences are visible in Precharge energy and Self Refresh energy. ALL consumes the most amount of Precharge and transition energy since it turns on all ranks at each file-I/O and the ranks remain in Precharge state during the entire I/O. The energy overheads of the other predictors are proportional to their prediction coverage as shown in Figure 4.4. Single-rank predictors: MRA, F-MRA, F-MFA, and F-DRD, each reduce ALL’s Precharge energy by 46%, 57%, 11% and 10%, respectively, with only a small increase in Self Refresh energy. Finally, they respectively yield a 33%, 37%, 21% and 21% reduction in total energy consumption, as compared to ALL. However, energy efficiency is achieved at a cost of higher delays as shown in Figure 3.5. Adaptive predictors AF-MRA, AF-MFA, and AF-DRD reduce delays more aggressively by turning more ranks on and consume on average 64%, 33%, 29% more Precharge energy than the single-rank predictors.

An interesting scenario occurs in TPC-H and Multi1 where the single-rank predictors F-MFA and F-DRD consume more Precharge energy than the other mechanisms. The explanation is twofold. First, the 37% coverage for F-MFA and F-DRD is the lowest, incurring the most on-demand transitions delayed until the physical memory accesses. Second, the average I/O size in TPC-H is around 66 bytes which only takes 9.8 ns for DDR2-800, and as a result, the 500 ns transition delay incurred by a misprediction prolongs the I/O time by 51 times. Therefore, during the severely prolonged I/O period, wrongly transitioned ranks consume higher Precharge energy and other ranks consume more Self Refresh energy, resulting in much higher total energy consumption.

The cases of TPC-H and Multi1 illustrate the negative impact of delays not only on performance but also on energy savings. We can expect similar behavior from
other power management mechanisms that sacrifice performance for energy savings
of a single component, but may cause the entire system to consume more energy.
Therefore, it is critical for power management mechanisms to focus on reducing
delays to minimize the impact of performance degradation on the overall energy
efficiency.

3.3.4 Energy Reduction with Early-Turnoff

The energy consumed by wrongly transitioned ranks can be reduced by turning them
off earlier, as described in Section 3.1.5. Figure 3.7 compares the effectiveness of
this optimization for adaptive predictors. Each mechanism with early-turnoff has a
suffix et, and its energy consumption is still normalized to ISR. Significant benefit is
achieved from early-turnoff optimization that eliminates Precharge energy consumed
by unused ranks. Subsequently, the early-turnoff technique reduces the total energy
consumption by 27%, on average, for all mechanisms shown in Figure 3.7. The best
mechanism with early-turnoff, AF-MRA-et, only consumes 3% more energy than
ISR when only considering memory energy. The energy efficiency would be actually
better than ISR if the entire system energy was considered.
3.3.5 Larger Memory Systems

We have evaluated our mechanisms in the context of a 4 GByte memory with 4 ranks. To extrapolate our findings to larger memory systems, we increase the number of ranks in the system to 8 and 16. We pick TPC-C and show the impact of rank increase in Figure 3.8, other applications behave similarly. The mechanisms evaluated here are all equipped with the early-turnoff technique to reduce energy consumption. The goal of adaptive mechanisms is to maintain the desired coverage Figure 3.8(a) that the coverage remains at 95%, as the number of ranks in the system increases. All single-rank predictors encounter reduction in rank transition coverage rate, and the most affected one is F-DRD-et. Coverage in F-DRD-et suffers because data blocks are becoming more fragmented with more ranks being available. As a result, coverage in F-DRD-et drops steadily from 55% at 4 ranks to 29% at 16 ranks.

The number of wrong transitions as shown in Figure 3.8(b) increases in the same rate as the coverage dropping rate when the number of ranks raises. Adaptive predictors however, are not degraded by the increased number of ranks, since the global preset coverage rate simply forces each predictor to turn on more ranks at a file-I/O to guarantee the 95% coverage rate all the time. Subsequently, the wrong transitions increases rapidly by up to 3 times when rank size raises from 4 ranks to 16 ranks.

Figure 3.8(c) shows the file-I/O time of each predictor normalized to I/O time with no delay, and the delay is shown above 1.0. Due to the higher number of missed transitions by the single-rank predictors, more delays are incurred resulting in performance degradation rising from 12% at 4 ranks to 16% at 16 ranks. Adaptive predictors desired coverage and as result maintain the low delay with less than 1% change for all different rank sizes. Energy savings also experience less than 1% change for all studied predictors. We can conclude that our mechanisms work very efficiently in terms of energy and performance under various rank sizes.

Figure 3.8(d) shows the energy consumption of physical memory during file-I/Os. The energy consumption at each rank size is normalized to ISR’ energy consumption at the same rank size which is shown as 1.0. Due to the early-turn off optimization,
all unused ranks that were wrongly turned on by each predictor are turned off. Despite of the significantly more wrong transitions of adaptive predictors at 16 ranks, unnecessary energy consumption is successfully eliminated by early-turnoff, and as a result, the energy consumption of each predictor stays fairly consistent compared to ISR at the same rank size. Also because of the finer granularity, the energy consumption of each predictor is reduced gradually. At 16 ranks, all predictors even achieve better energy efficiency than ISR, consuming 1% less energy due to the shorter runtime.
Figure 3.8: Results of various predictors under different rank size for TPC-C.
In interactive environment, there exists a strong correlation between user interactions and the CPU workload required by the triggered tasks. The prediction accuracy of a DVS algorithm for interactive applications is therefore dependent on the information gathered at application runtime and the granularity of the captured context of interaction and execution. For example, Figure 4.1 shows the main toolbar of the GNU Image Manipulation Program (GIMP). When interaction context includes only the main window components [28], as shown in Figure 4.1A, individual interactive elements are aliased to the containing region and the individual tasks initiated by these elements are therefore indistinguishable to the predictor. On the other hand, the fine-grained interaction context shown in Figure 4.1B allows the predictor to correlate the CPU demand level of each interactive element. The fine-grained context resolution results in lower classification variability and highly accurate predictions.

The variability in the CPU demand following interactions on individual UI components is shown in Table 4.1. We have selected several popular features that a user may invoke in GIMP. All of the interactions belong to a single menu region and are classified into the same interaction group as shown in Figure 4.1A. For example,

<table>
<thead>
<tr>
<th>Interactive Element</th>
<th>Window Region</th>
<th>Elem. Mean</th>
<th>Elem. StdDev</th>
<th>Region StdDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Margin</td>
<td>menu</td>
<td>35</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Help</td>
<td>menu</td>
<td>207</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>ZoomOut</td>
<td>menu</td>
<td>249</td>
<td>59</td>
<td>303</td>
</tr>
<tr>
<td>Rotate</td>
<td>menu</td>
<td>570</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>ColorInvert</td>
<td>menu</td>
<td>934</td>
<td>158</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Statistics of CPU demand (Million Cycles) for mouse click events in GIMP.
Figure 4.1: GUI menus in GIMP. A: Menu and toolbar area are distinguishable. B: Individual elements are distinguishable.

classifying a margin adjustment (35 million cycles) and the inverting of image colors (934 million cycles) into a single category results in a high standard deviation (303 million cycles). However, when the menu region is broken down to its constituent elements, as shown in Figure 4.1B, we obtain a much lower standard deviation for the associated CPU demand.

We summarize the differences between categorizing tasks by individual interactive elements and categorizing tasks by coarse-grained window components as Lorch at el. proposed [28]. Table 4.2 shows the mean Coefficient of Variation (CV) of the task CPU demand for both task categorizing granularities. The CV is a normalized measure of dispersion obtained by the ratio of standard deviation to the mean. When tasks are distinguished only by the coarse context of window components, the mean CV is considerably high, indicating that the CPU demand is heavily dispersed for the aliased tasks. On the other hand, when tasks are distinguished by the exact interactive elements that initiated them, the CV of the tasks’ cycles is reduced by 74%.

4.1 IADVS design

Based on the above observations, we propose the Interaction-Aware Dynamic Voltage Scaling (IADVS) mechanism for CPU energy management, by using high-resolution capture of user interactions and task classification as described above. Previous approaches either don’t distinguish any types of user interactions [15] or
Table 4.2: Mean Coefficient of Variation (CV) of the tasks’ CPU demand.

<table>
<thead>
<tr>
<th>Application</th>
<th>Element Mean CV</th>
<th>Region Mean CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbiWord</td>
<td>0.5591</td>
<td>1.8325</td>
</tr>
<tr>
<td>Gnumeric</td>
<td>0.5567</td>
<td>2.9560</td>
</tr>
<tr>
<td>Scigraph</td>
<td>0.5642</td>
<td>1.7407</td>
</tr>
<tr>
<td>Eclipse</td>
<td>0.6048</td>
<td>2.3528</td>
</tr>
<tr>
<td>GIMP</td>
<td>0.3179</td>
<td>1.1458</td>
</tr>
<tr>
<td>LiVES</td>
<td>0.3649</td>
<td>1.5038</td>
</tr>
</tbody>
</table>

Figure 4.2: IADVS design architecture.

at most distinguish them at coarse granularity [28]. We exploit the fine-grained correlation between user interaction and task CPU demand to transparently transition the processor to a power/performance state that meets the task’s performance demand while saving energy. The following are the components of the proposed IADVS mechanism: (1) High-detail and low-overhead monitoring mechanism for detecting the tasks triggered by user interactions; (2) Low-overhead correlation and classification of interactions and the triggered CPU tasks; (3) Fully online training and prediction for accurately determining the desired CPU frequency for the upcoming tasks; (4) Support for multi-core CPUs.

4.1.1 Distinguishing UI Interactions

The IADVS system relies on accurate capture and categorization of individual user interactions. Refined from the previous fine-grained capture mechanisms developed
for energy management for wireless network cards and hard disks [7, 8], IADVS utilizes a monitoring layer (X Monitor shown in Figure 5.3) between the X Window Server and applications in Linux, and utilizes the GUI window structure to identify each interactive component (such as each button or menu component) with a unique integer ID. As designed, the interactive applications connect to the X Monitor, which in turn passes unmodified interaction data and window change requests between the X Server and the client applications. The unique IDs used by IADVS are generated from the interactive element’s enclosing window and the element’s position in the application’s window tree as well as its relative position within the containing window. Subsequent interactions with a particular interactive element generate the same interaction ID, and the same categorization for the task to follow.

We further augment the X Monitor to collect detailed keystroke information from the keyboard. Keystrokes are captured and distinguished by a unique identifying number for each key and key combination on the keyboard. In addition, we also record the type of the UI event for both keyboard and mouse, such as a button-press or button-release. Mouse clicks are also distinguished as left, right, or middle button press, or a double click of any of the mouse buttons.

Mouse-driven interactions with the GUI uniquely identify the application with its context, due to the structure of each application’s GUI layout. Keyboard IDs are generic and not unique to an application. However, combinations of mouse interaction IDs and keyboard interaction IDs result in a unique context description for a given application. The combination of the capture mechanisms forms a highly accurate system whose operation is entirely transparent to the user.

4.1.2 Detecting UI-triggered Tasks

The fundamental issue in energy management for CPUs is the frequency of power state switching. Frequent switching can often carry significant transition overheads, while infrequent switches may result in disparate tasks being executed at a CPU frequency that either do not meet performance demand or do not provide the desired energy savings. Subsequently, we select tasks triggered by user interactions as the
main switching granularity. We define a task to be the sequence of operations by one or more threads that accomplish the same goal [15, 28].

Specifically, when a thread processing an event causes a new event to occur, the new event is said to be dependent on the first event and is counted as a part of the current task. A trigger event is the first event in a series of events that is not dependent on any other events. Finally, a task is defined as the collection of all CPU processing, including its trigger event and all of the dependent events. The UI monitoring system identifies the application from user interactions, isolating the tasks initiated by the running application shortly following an interactive event.

We define UI-triggered tasks as those tasks that are preceded by UI events which occur when the UI controller (the X Window server in Linux) receives a mouse click or a keystroke. UI-triggered tasks include not only the handling of the device interrupt, but also the processing required to respond to the UI event. For example, when a user clicks the “blur” button in GIMP, the triggered task includes both the processing of the associated application functionality as well as the processing of the mouse interrupt event. The task ends when the operating system’s idle process (swapper process in Linux) begins running, but the UI-triggered task is not blocked by I/O, or when the application receives a new user interface event [28]. It should be noted that the time a task spends waiting for I/O is not included in the total task duration in the analyses that follow. Likewise, task duration excludes preemption periods due to background daemons, such as the Name Server or NFS, since these processes are not dependent on the task that is being monitored.

4.1.3 Correlation Mechanisms

When a UI event initiates a task, IADVS begins monitoring the system activities, recording the duration of CPU time for each relevant process. Once task completion is detected, the task’s CPU demand is represented by the task length at the CPU’s maximum frequency, which is computed as the sum of the processing time of each event during the task execution. To accurately measure the length of a task, we utilize the high-resolution Time Stamp Counter CPU register. At task completion,
the corresponding entry in the prediction table is updated as shown in Figure 5.3. IADVS utilizes a hash table indexed by user interaction IDs as the prediction table. Prediction table entries consist of three fields: the interaction ID of a UI event, a weighted sum $SUM$ of previous task lengths, and a weighted count $COUNT$ of the observed task instances triggered by the same ID.

We use the Aged-$\alpha$ method to record the history of the past tasks’ CPU demand triggered by the same interaction ID. Aged-$\alpha$ utilizes all past tasks, with the $k$th most recent having weight $\alpha^k$. Hence, it emphasizes the most recent tasks, permitting a quicker response to changes in the task’s CPU demand, while smoothing anomalous task behavior at the same time. For each update with the most recent task’s length $L$, both $SUM$ and $COUNT$ are calculated using their previous values as follows:

$$SUM = \alpha \times SUM + L$$

$$COUNT = \alpha \times COUNT + 1$$

Therefore, IADVS need only keep $SUM$ and $COUNT$ in the table entry to record the CPU demand of all past tasks.

4.1.4 Predicting CPU Power Modes

Each time a UI event occurs, IADVS performs a prediction table lookup using the captured interaction ID. The lookup results in two possible outcomes: (1) the entry is found with the weighted sum ($SUM$) of previous task lengths and weighted count ($COUNT$) of previous tasks; or (2) the entry is not found, indicating that the interaction ID has been seen for the first time. If the entry is found, IADVS first predicts the task length for the upcoming task as the average task length $L_{avg}$, simply calculated as $SUM$ divided by $COUNT$. The resulting frequency $F$ remains constant for the duration of the task, or until the the perception threshold is reached, and is calculated as follows:

$$F = (L_{avg} / PerceptionThreshold) \times maxFrequency$$
The CPU’s frequency setting that best matches $F$ (equal to or just higher) is predicted, and the frequency and voltage of the CPU are set accordingly. If $F$ is greater than maximum available frequency, it indicates that the earlier tasks triggered by the same interaction ID are usually longer than the perception threshold, and thus IADVS selects the maximum available frequency to run the task. To minimize performance degradation due to mispredictions, where a task continues past the perception threshold at a frequency lower than maximum, IADVS sets the CPU to the maximum frequency immediately upon reaching the perception threshold deadline.

When an interaction ID is first seen by IADVS, a corresponding entry in the prediction table does not yet exist, requiring a heuristic for setting the CPU frequency for the upcoming task. Selecting the maximum frequency eliminates performance degradation during initial training, at the cost of a potential, but relatively small, increase in energy consumption. On the other hand, selecting the minimum frequency prevents the increase in energy consumption, while introducing the potential for higher delays. Since one of the goals of IADVS is to minimize the impact of energy management on the user, we select the CPU’s maximum frequency during initial training, eliminating delays that may otherwise be apparent to the user.

4.1.5 Managing Multi-core CPUs

Modern systems utilize multi-core CPUs to provide more processing power for execution of concurrent application or processes. Subsequently, any CPU management mechanisms have to be able to work for multi-core CPUs. Current multi-core CPUs allow per-core frequency settings, but voltage settings affect the entire chip. The CPU voltage is set to match the highest frequency setting of any of the cores. For example, if one core is running at 3GHz the chip’s voltage is set to support that frequency. As a result, energy savings are limited since the majority of the savings come from reduced voltage levels. While voltage regulators remain external to the chip, adjusting core voltages independently will not be possible. However, on-chip regulators considerably increase chip complexity, so it is more reasonable to expect
voltage adjustments to be made to banks of cores.

IADVS design is independent of chip design since it sets the frequency only, and the voltage levels are determined by the maximum frequency that one of the cores is operating at. The collection of a task’s execution context happens on a per-process basis and is not tied to any specific core. The task frequency settings are tied to the particular task and if this task is rescheduled on another core the frequency level is set accordingly. Therefore, the simplicity of the IADVS design allows it to work on multi-core systems without special considerations or redesign.

4.2 Methodology

To evaluate the proposed IADVS mechanism, we developed a trace-driven simulation of IADVS and the following DVS mechanisms:

- **STEP.** An interval-based mechanism which runs each task by starting at the minimum frequency, and stepping up the next available frequency following a fixed time interval. A 10ms interval length is chosen to meet the average task demand.

- **APS.** A task-based mechanism that predicts the upcoming CPU demand for all UI-triggered tasks, but does not distinguish any of them [15].

- **SPACE.** A task-based mechanism which separates UI-triggered tasks by the UI events based on the key pressed and the type of window component a mouse event occurred in. SPACE is essentially the PACE-CA mechanism [28], but simplified to apply a constant frequency for each task.

- **ORACLE.** A task-based mechanism that utilizes future knowledge to select the optimal operating frequency for the task, fitting it to the deadline or running at maximum frequency if the task is longer than the deadline.

We assume that each DVS mechanism transitions the CPU to the minimum frequency immediately following a task, when the CPU begins idling.

The traces used in the simulation were collected using a modified Linux kernel (2.6.27.5-117) running on an AMD Phenom II X4 940 with 4GB RAM. All traces
<table>
<thead>
<tr>
<th>Benchmark Application</th>
<th>Trace Length (hrs)</th>
<th># of User Interface Tasks</th>
<th># of Unique Interaction ID</th>
<th># of Unique Mouse Clicks</th>
<th># of Unique Keyboard Strokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbiWord</td>
<td>4.50</td>
<td>18227</td>
<td>286</td>
<td>195</td>
<td>91</td>
</tr>
<tr>
<td>Gnumeric</td>
<td>4.56</td>
<td>13844</td>
<td>479</td>
<td>382</td>
<td>97</td>
</tr>
<tr>
<td>Scigraph</td>
<td>2.43</td>
<td>3795</td>
<td>261</td>
<td>248</td>
<td>13</td>
</tr>
<tr>
<td>Eclipse</td>
<td>5.35</td>
<td>21933</td>
<td>361</td>
<td>266</td>
<td>95</td>
</tr>
<tr>
<td>GIMP</td>
<td>2.46</td>
<td>3861</td>
<td>261</td>
<td>220</td>
<td>41</td>
</tr>
<tr>
<td>LiVES</td>
<td>1.76</td>
<td>1536</td>
<td>130</td>
<td>119</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 4.3: Application trace statistics.

contain data from a large number of typical-usage sessions of a single user in the GNOME 2.20.1 environment. Application traces are composed of two components: the user interface event trace, and the process activity trace. Process activity traces are collected by using Linux Trace Toolkit (LTTng) that logs program execution details from a patched Linux kernel. The traces include all significant OS events: system calls, thread swap events, disk/network I/Os, and process information. These events can provide in-depth knowledge about a running process, including when context switches occurred, or how much time a process’s task spent executing or waiting for I/O. Based on the process information and the ordered event timestamps, our simulator can rebuild the dynamic execution progress of the traced application and analyze the effects of applying DVS.

We use six applications commonly executed on desktop or mobile systems: AbiWord—a word processing application, Gnumeric—a spreadsheet application, Scigraph—scientific data plotting software, Eclipse—an Integrated Development Environment for C/C++ and Java development, GIMP—image processing software, and LiVES—an integrated video editing and playback toolkit.

Trace details, such as the total length of the trace duration, the number of UI-triggered tasks, and number of unique interaction IDs, are included in Table 5.1. Also included are the counts of unique interaction IDs broken down into mouse clicks and keyboard strokes, which serve as an indicator of the GUI complexity for each application.

Through experimentation with the application traces, we found that when using the “Aged-α” method for computing a weighted mean, the weight α of 0.95 gen-
erates the highest prediction accuracy and thus the best overall results. We also use a perception threshold of 100ms as the deadline for UI-triggered tasks in our experiments.

4.2.1 Processor Model

We used the AMD Phenom II X4 desktop CPU [1] to study the implementation of the proposed mechanisms (the CPU specifications are summarized in Table 4.4 in Section 4.4). Similarly to the previous findings [15, 28, 29], experiments are constrained by the small number of available CPU operating states. The four available power states are insufficient to efficiently match the CPU performance to task demand. Therefore, we simulated a custom CPU model based on AMD Phenom II with the frequency range from 800MHz to 3.0GHz, supporting the intermediate frequencies in 250MHz increments. We’ll refer to this model as the Full Frequency Range CPU (FFRCPU). With FFRCPU, we assume that voltage, $V$, is linearly proportional to the corresponding frequency $f$:

$$V = 1.25 \times 10^{-4} f + 1.05$$

We further leverage the property of power consumption $P$ being proportional to $V^2 f$ and assume that the CPU consumes 40.8W at the maximum frequency which is the same as the measured power consumption for AMD Phenom II. We will use FFRCPU to evaluate the described DVS mechanisms in the following section.

Figure 4.3 shows the distribution of all UI-triggered tasks based on the optimal CPU frequency required to meet the deadline for the FFRCPU model. It is clear that the processing demand of tasks varies across applications. In AbiWord, Gnumeric and Eclipse almost 90% of the tasks are low-demand workloads easily meeting the deadline at frequencies lower than 1.55GHz. This is due to the user spending most of the time typing, editing, and formatting. On the other hand, due to graphics and video processing, GIMP and LiVES exhibit high-demand CPU workloads, where over 50% of the tasks require an operating frequency of 1.80GHz or above. Lastly, Scigraph’s task distribution is balanced, exhibiting both high and low CPU demands.
in similar measure. The task’s CPU demand distribution provides useful insights into the remaining analysis.

4.3 Evaluation

4.3.1 Accuracy

Upcoming task demand prediction accuracy is the key metric that largely determines the actual performance and energy efficiency of task-based DVS mechanisms. We compare the prediction accuracy of different task-based mechanisms to the ORACLE mechanism, which has future knowledge of upcoming task length. The prediction
distance from the predicted frequency to the optimal frequency is the number of frequency settings between them. A distance of 0 means that the frequency is correctly predicted, i.e. the same as ORACLE’s prediction. Distances other than 0 indicate that the frequency is mispredicted and not optimal. Since there are 10 different frequency settings in FFRCPU, the range of distance values is between -9 and 9, inclusive. For instance, if a frequency of 800MHz is predicted but the optimal is 1.3GHz, the prediction distance is -2. A negative distance indicates that the predicted frequency is lower than required, resulting in a missed deadline and the associated delay. A positive distance indicates that the frequency is higher than required, leading to excessive energy use.

Figure 4.4 shows the aggregate prediction distance for each task-based mechanism over the six applications. Intuitively, the best predictor has the largest fraction of tasks at the optimal frequency. IADVS predicts the exact frequency for 38% of the tasks, resulting in accuracy that is 36% and 34% higher than APS and SPACE, respectively. In addition, Figure 4.4 shows that IADVS contains the fewest tasks in both the positive and negative distance ranges, meaning that it mispredicts the associated frequencies less often than the others. IADVS, therefore, has the potential to save the most energy while incurring the lowest delays among the three mechanisms.

Figure 4.5 shows the absolute prediction error of each predictor for each work-
Figure 4.6: Runtime normalized to ORACLE.

load. The absolute error sums up the absolute values of all the prediction distances in Figure 4.4. We use APS’s prediction error as the basis for normalizing the errors of the other mechanisms. The prediction error for each mechanism is further broken down into four parts: the cold key and mouse error due to the default frequency selections for the first seen keystroke/mouse-click tasks, and the normal key/mouse errors due to the mispredictions for recurring keystroke/mouse-click tasks. APS generates the highest prediction error since the UI-triggered tasks are indistinguishable from one another. SPACE reduces APS’s key error by 75% while retaining a comparable error rate for mouse clicks. IADVS improves SPACE by distinguishing the specific interactive components, reducing SPACE’s mouse error by 54%. IADVS’s higher granularity generates more task categories and thus results in a higher cold prediction error. Despite this cold training overhead, IADVS significantly reduces overall prediction errors by 47% from APS and 37% from SPACE.

4.3.2 Delay

Performance is an important factor for evaluating the effects of DVS mechanisms on interactive applications, since users do not tolerate excessive delays for the UI-triggered tasks. Figure 5.8 shows the total runtime for each application and for each mechanism normalized to ORACLE’s. Any task that runs longer than its runtime in ORACLE exposes additional delays to the users.
Frequency stepping from the minimum in STEP results in long delays due to the time taken to ramp up to the maximum frequency for tasks longer than the perception threshold, which dominate in GIMP and LiVES. On average, STEP incurs 16%, 42% and 81% more delay than APS, SPACE, and IADVS, respectively. The best performance of IADVS mirrors its highest accuracy in predicting the CPU frequency, as shown in Figure 4.4. IADVS has the fewest tasks mispredicted with a lower frequency than is required (the negative range in Figure 4.4). In Scigraph, IADVS achieves the most improvement with 54% and 52% less delay than APS and SPACE, respectively. Note that Scigraph’s interaction patterns frequently interleave short-duration and long-duration tasks. Lack of categorization of user interactions, as in APS, or categorization with coarse resolution, as in SPACE, fails to correctly classify these tasks. In AbiWord, Gnumeric and Eclipse, a few long-duration mouse-click tasks are mixed with the majority short-duration keystroke tasks. IADVS accurately recognizes these task patterns and predicts the appropriate frequency for long-duration tasks so that it reduces APS’s delay by 37%. SPACE, by distinguishing mouse-clicks and keystrokes, performs relatively well, incurring only 17% more delay than IADVS. In GIMP and LiVES, the majority of long-duration tasks accompany with occasional short-duration tasks. Lacking sufficient user interaction details, APS and SPACE only recognize the long-duration pattern, and aggressively apply higher CPU frequency even for short-duration tasks. Their delays are relatively smaller, but at the cost of increased energy consumption as we will see in the next section. IADVS in these two applications performs comparably to APS and SPACE, but saves more energy.

4.3.3 Energy

Figure 5.7 compares the energy consumption of the studied mechanisms, normalized to ORACLE’s energy consumption. Each DVS mechanism compared here is made to transition the CPU to the minimum frequency immediately when the CPU begins idling. Our focus is on energy consumed by the CPU when it is engaged in performing some task related to the application being interacted with. Subsequently,
the energy consumed during CPU idling is the same across all mechanisms, and is excluded from the total energy consumption in Figure 5.7.

IADVS achieves the best energy efficiency among task-based mechanisms, reducing the energy consumption of APS and SPACE by 2% and 3% on average. In best cases of \textit{GIMP} and \textit{LiVES}, where APS and SPACE over-predict the CPU frequency for short-duration tasks, IADVS reduces energy consumption by 5% and 4% respectively. STEP consumes comparable energy as IADVS, due to its starting from minimum frequency and the gradual stepping up. However, STEP trades energy efficiency for a significantly degraded execution performance, as shown in Figure 5.8, which may result in increased energy consumption by other components since the entire system has to stay on longer.

4.3.4 Energy-Delay\(^2\) Product

To simultaneously compare both energy and performance we require a metric that combines both. Energy-delay product (ED) is one such metric, but since \(E \propto V^2\) and \(T \propto 1/V\), ED product can be minimized by simply reducing the voltage at the expense of performance. A better metric is the energy-delay\(^2\) (ED\(^2\)) product, which to the first order is independent of voltage. A DVS mechanism with a lower ED\(^2\) product provides higher performance than another at equivalent energy levels, or consumes less energy at the same performance level. Figure 4.8 shows the ED\(^2\)
product for studied mechanisms normalized to ORACLE. Idling energy and time are excluded from the execution’s total energy consumption and runtime.

We observe that IADVS achieves the best improvement of the ED^2 product, outperforming STEP, APS and SPACE by an average of 6%, 5%, and 4%, respectively. STEP has the longest execution time and thus cannot significantly offset its delays by lowering energy consumption. It is not surprising to see that STEP performs the worst, especially in GIMP and LiVES where frequency adjustments for computationally intensive tasks occur gradually, introducing delays. SPACE improves APS by 0.1% on average, implying that distinguishing tasks by coarse categories of window components is not adequate. In the best case, Scigraph, IADVS achieves as much as 9% and 10% improvement over APS and SPACE, respectively.

In summary, IADVS achieves the best execution performance at the cost of the lowest energy. Therefore, we conclude that increasing the resolution of GUI event capture and fine-grained classification of tasks is worthwhile, since higher prediction accuracy translates to a significantly lower energy-delay^2 product.

4.3.5 The Need for FFRCPU

As discussed in section 4.2.1, our evaluation of DVS mechanisms included a realistic CPU model, whose available frequency settings proved to be too limited for an effective DVS mechanism implementation. To illustrate this, Figure 4.9 shows the
Figure 4.9: Energy-delay$^2$ product for AMD CPU normalized to ORACLE in FFR-CPU.

$ED^2$ product for each DVS mechanism as applied to the real CPU model of AMD Phenom II X4, shown in Table 4.4. Here, the mechanism names include the prefix AMD to distinguish them from their counterparts in the FFRCPU model used thus far. The results are normalized to the FFRCPU’s ORACLE mechanism. Comparing Figure 4.8 and Figure 4.9 clearly shows that the limited number of frequency settings in the AMD model prevents the efficient matching of CPU frequency to tasks and as a result, the ED$^2$ products are degraded for all mechanisms. We note that even with less frequency settings, IADVS still outperforms the other mechanisms. Comparing the ORACLE mechanism using the FFRCPU and the AMD model, we find a 4% increase in ED$^2$ product using the AMD model. Therefore, it clear that if we want to take full advantage of the proposed IADVS as well as the other existing DVS mechanisms, we need wider frequency-voltage scaling than is currently available in the CPUs.

4.4 Implementation

In this section, we compare the efficiency of the Linux Performance governor (MAX), which essentially races-to-idle: using the maximum frequency for each task and switching to minimum when idle, with the implementation of IADVS mechanism. We first describe implementation and measurement details, followed by the experi-
mental results.

4.4.1 Implementation Details

The experiments were conducted on a desktop computer with Fedora 10 64-bit kernel (2.6.27.5-117), running on a AMD Phenom II X4 940 quad-core CPU, and 2x2GB DDR2-1066 RAM. The Phenom II CPU comes with the latest Cool’n’Quiet (C’n’Q) 3.0 power saving technology, that provides four performance states (p-state) for each CPU core. The frequency-voltage transition for one core is performed through the p-state transition of that core. Table 4.4 shows the combination of the operating voltage and frequency for each of the evaluated p-states. When the processor is not idling, it is in one of the described p-states, on the other hand, when the processor is idling it enters one of the idle state (C1E state in our study).

As shown in Figure 4.10, the power consumption of the CPU was measured using a NI PCI 6230 DAQ recording the voltage drop across a 0.01Ω resistor inserted in the CPU power supply. System power consumption was measured with a WattsUp power meter. The collected measurements are shown in Table 4.4.

4.4.2 Results

Figure 4.11 shows the measured power variation for Scigraph over a selected 3-second execution period under the two DVS mechanisms. When MAX was used shown in
Table 4.4: Phenom II X4 940 p-states with the associated CPU voltages (VDD), current operating frequencies (COF) and the power consumptions (in Watts).

| P0 | 1.425 | 3000 | 29.2 | 40.8 | 110.7 | 124.2 |
| P1 | 1.325 | 2300 | 20.8 | 28.0 | 99.8  | 109.1 |
| P2 | 1.225 | 1800 | 15.3 | 20.3 | 92.6  | 99.2  |
| P3 | 1.150 | 800  | 10.8 | 12.8 | 86.6  | 89.3  |

upper figure, the CPU performance always transitioned to maximum for each task, and thus always raised up to maximum power consumption. Furthermore we see more switches that transition the CPU between two extreme frequency states. Those transitions add some switching delays as well as energy consumed during the transition. IADVS has fewer transitions than MAX, eliminating the switching overheads as well as selecting more intermittent frequencies to better match the task’s demand. We measured transition overheads of performing a p-state transition on the AMD Phenom II CPU. The observed overhead is around 50us, from the time when the DVS mechanism sent a transition request to the time when the CPU completed the transition. For each task, MAX has to transition the p-state twice: one to maximum to run the task and the other to minimum when idling. IADVS, due to its ability to accurately predict the task demand and the corresponding performance level, uses much less transitions, which in turn saves more energy and transition delay. Finally, higher utilization of lower frequency levels in IADVS provides higher energy savings than MAX without extending the application execution time.

Table 4.5 shows the energy consumed by the CPU. With MAX, the CPU consumed more energy simply due to the maximum power consumption at the maximum frequency and voltage. IADVS, by predicting the CPU performance rather than using the maximum, improves the energy efficiency of MAX significantly. IADVS saves on average, 6% of the energy from MAX. Note that the energy numbers presented in Table 4.5 also include the extra energy consumption due to the frequency-voltage transitions that are not related to the application execution. The extension in execu-
Figure 4.11: Power consumption in a selected 3-second period during the execution of *Scigraph* under different DVS mechanisms.

<table>
<thead>
<tr>
<th></th>
<th>CPU energy consumption (Joules)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AbiWord</td>
</tr>
<tr>
<td>MAX</td>
<td>43259.5</td>
</tr>
<tr>
<td>IADVS</td>
<td>40097.3</td>
</tr>
</tbody>
</table>

Table 4.5: Energy consumed by the CPU.

...tion time in case of IADVS was less than 1% for the studied applications; therefore we can expect that energy savings from the CPU will extend to the rest of the system.
CHAPTER 5

INTERACTION-AWARE MEMORY ENERGY MANAGEMENT

Current trends in providing larger on chip CPU caches result in main memory seeing fewer accesses from the CPU, which creates longer memory idle times. Subsequently, the majority of memory energy is consumed in the idle state. Energy consumption of main memory can be significantly reduced by transitioning memory devices to a low-power state during the idle periods. However, accessing memory in a low-power state incurs high transition latency, and as a result, degrades the system performance and may increase the overall energy consumption. Therefore, it is crucial to ensure that the associated performance degradation can be hidden behind the application execution and not exposed to users.

Fortunately, the full performance is usually not needed in interactive applications, since users are unable to perceive certain amount of short delays. We can exploit that observation in designing more aggressive energy management mechanisms. Prior studies in human-computer interaction have established the human perception threshold to be between 50-100ms, indicating that events with durations falling within this threshold are not perceived by the user [37]. Completing task execution earlier than the perception threshold is meaningless since the user will not notice this amount of time and cannot initiate tasks any faster. Therefore, any task shorter than the perception threshold can be potentially executed at a lower performance level, so that its execution time can be stretched up to, but not beyond the perception threshold. The resulting lower power consumption can improve energy efficiency while the user’s behavior is unaffected.

To account for all possible users and prevent any potential performance degradation, we assume the lower bound of 50ms as the perception threshold for all users. In the remainder of this chapter, we refer to any task finishing within 50ms as a short task and any task running longer than 50ms as a long task. A short task appears
Figure 5.1: Distribution of tasks with memory in high and low power states. Shaded bar represents the fraction of short tasks extended over 50ms, while shaded-crossed bar represents the fraction of long tasks extended by more than 50ms.

instantaneous to the user even if extended up to 50ms. A long task, however, is perceivable to the user even at the highest performance level. Since the user would not be able to perceive the time difference within 50ms, a long task can be safely prolonged by up to 50ms, assuring that the user’s think flow is not interrupted and the subsequent behavior is not affected [5, 33]. To take advantage of the allowed delays in interactive applications, we can potentially put the entire memory subsystem into a low-power state between memory operations and power it up upon a memory access request. The transition latency due to this on-demand power up may slow down a single memory access; nevertheless, from the task perspective, the user may not notice the aggregated delays, as long as the scaled task does not exceed the user’s perception threshold as discussed above.

Figure 5.1 examines the scenario of keeping memory in a low-power state (Low) between accesses for several interactive applications and compares it to the standard system that keeps memory in a high-power state (High). The tasks from each application are further classified into short tasks and long tasks. Keeping memory in the low-power state can extend task execution beyond the user’s perception threshold, as shown in Figure 5.1 by shaded area in each category. We observe that 93% of short tasks and 58% of long tasks stay within the user’s perception tolerance. The longer
tasks suffer more since they perform more memory operations and as a result, expose more transition delays. At this point, we can draw two significant observations: (1) there is a tremendous opportunity to save energy within a running application by keeping memory in a low-power state; and (2) some tasks have to be executed with memory in the high performance state, otherwise the degradation would be noticed by the user. The observations justify the need for an intelligent mechanism that is able to accurately identify memory intensive tasks from the majority of low-demand tasks that can be executed at low memory performance.

5.1 IAMEM design

The majority of tasks in interactive environments are initiated directly by users and the performance demand within an application exhibits a strong correlation to User Interactions (UIs) with the application [7, 8]. Similar to IADVS proposed in the previous chapter, memory behavior and performance requirement in interactive system also exhibits strong correlation with the user interaction. We therefore propose Interaction-Aware Memory Energy Management (IAMEM) for entire memory space in interactive systems. IAMEM will transparently exploit UI events to speculate the desired performance, and dynamically manage the memory power states to meet the task demand. Subsequently, we will discuss the following components in this section: (1) Unified energy management mechanisms that address all types of accesses to physical memory; (2) High-detail and low-overhead monitoring and detection of tasks triggered by user interactions; (3) Accurate classification and correlation of tasks and the associated processing demand; (4) Online training and prediction for determining the desired memory power for upcoming tasks; and (5) Optimizations to prevent perceivable performance degradation.

5.1.1 Memory Space

Physical memory in modern operating systems is divided into three categories, as shown in Figure 5.2: (1) kernel space that is strictly reserved for the OS kernel,
its data structures, device drivers, etc.; (2) the buffer cache for caching previously accessed disk blocks to improve the file system performance; and (3) user space that is allocated as Virtual Memory (VM) for user processes. The majority of memory space is dynamically allocated to the buffer cache and virtual memory of running processes, based on the current demand for each type of memory.

As discussed in Chapter refsec:buffer, memory ranks used for the buffer cache can be efficiently managed in server environments by hiding power-state transition overheads behind the kernel processing time. While the mechanism worked well in server workloads where the buffer cache occupies large space spanning several ranks, it has limited applicability for interactive applications where the buffer cache occupies smaller space and usually shares the rank with the kernel data structures. Subsequently, upon a first kernel memory access, the rank is powered up making large portion of the buffer cache accessible without further delays. Even if the buffer cache occupies several ranks, interactive applications, in general, put lesser pressure on the buffer cache than server applications, such that only a small fraction of overall accesses may require powering up additional ranks. Therefore, we consider memory space occupied by the kernel data structures and the buffer cache as a single kernel memory space and do not distinguish them further. Subsequently, we propose unified energy management mechanisms that manage all memory spaces based on user interaction patterns to guarantee user-perceivable performance while maximizing energy savings.

5.1.2 UI and Task Correlation

IAMEM classifies tasks by the individual interaction IDs of the triggering UI events [7, 8]. To predict the memory power demand for each task category, IAMEM
utilizes a prediction table implemented as a hash table indexed by the interaction IDs. Once the completion of a task is detected, the Aged-α statistical method is used to update and record the task processing demand described by the execution time and the number of memory references. Figure 5.3 illustrates the prediction table with the following variables for each interaction ID: (1) a weighted sum of all previous tasks’ computation time $Time$; (2) a weighted sum of all previous tasks’ memory references $MemoryRefs$; and (3) a weighted count of all observed task instances $Count$. For the most recent task with computation time $T$ and memory references $M$, those three table variables are updated as follows with a predefined weight $\alpha$ ($\alpha <= 1$):

$$Time = \alpha \ast Time + T$$

$$MemoryRefs = \alpha \ast MemoryRefs + M$$

$$Count = \alpha \ast Count + 1$$

Note that $T$ is recorded as the actual computation time with memory in the active state. When memory transitions occur, the time spent in power state transitions should be deducted from the actually monitored time.

5.1.3 Power State Prediction

Each time a UI event occurs, IAMEM performs a table lookup using the captured interaction ID. It first predicts the incoming task’s processing demand as the average
\[ T_{\text{avg}} = \frac{\text{Time}}{\text{Count}}; \]
\[ M_{\text{avg}} = \frac{\text{MemoryRefs}}{\text{Count}}; \]

\begin{align*}
\text{if } (T_{\text{avg}} \leq 50\text{ms}) & \quad D = 50\text{ms}; \\
\text{else} & \quad D = T_{\text{avg}} + 50\text{ms}; \\
\text{if } (T_{\text{avg}} + M_{\text{avg}} \times L_{sr} < D) & \quad PS = SR; \\
\text{else if } (T_{\text{avg}} + M_{\text{avg}} \times L_{pd} < D) & \quad PS = PD; \\
\text{else} & \quad PS = PRE;
\end{align*}

Figure 5.4: The power-state prediction algorithm for an upcoming task. \( L_{pd} \) and \( L_{sr} \) are the transition latencies from PD and SR state to PRE state, respectively.

of the retrieved demand history. To maintain the user-perceivable performance, an appropriate deadline \( D \) is selected, which is either 50ms for short tasks, or the task’s computation time, with memory in the active state, plus 50ms for long tasks. Based on this deadline, IAMEM calculates \( PS \), the lowest possible power state of a memory rank, to fit the task execution within the deadline when all predicted memory accesses encounter power state transitions. The prediction algorithm is shown in Figure 5.4. The resulting \( PS \) gives us the needed power state to accomplish the task before the deadline and can be any of the three states: Precharge (PRE), Powerdown (PD), or Self Refresh (SR). Finally, \( PS \) is set in the memory controller which transitions a rank to the predicted power state after any access to that rank finishes. Therefore, a rank is transitioned to PRE state upon the first access request and then transitioned to \( PS \) after that access completes. Once a task completes and the CPU enters an idle state, all ranks are set to SR state and will remain in that state until a new memory request arrives.

Selection of PRE state indicates that the running task cannot tolerate any transition delays to finish before the deadline and thus the task must be executed at
the highest performance. Selection of PD or SR states, on the other hand, indicates that the running task can tolerate power-state transition delays associated with the selected power state while still being able to meet the deadline. We should note that the calculations in Figure 5.4 assume the worst case scenario where memory I/Os are not clustered but arrive one at a time, since we are unable to capture the exact access patterns but only the total number of accesses during the task execution. Subsequently, the calculated delays are the maximum predicted delays the task may encounter, minimizing the possibility that the tasks would continue past the deadline. If memory accesses are bursty, arriving together, the actual exposed delays will be lower.

To avoid exposing potentially large delays to the users while still providing some energy savings, we utilize PD state during the training of the predictor. Our experiments show that almost all deadlines can be met with memory in PD state, which is also much more energy efficient than PRE state. Subsequently, IAMEM uses PD state when the entry in the prediction table is not found. Finally, interaction IDs are unique across applications and thus can be maintained in a single table in the kernel across executions for all applications, further minimizing the impact of training.

5.1.4 Improving Prediction Granularity

IAMEM prediction mechanism described earlier utilizes only a single number of memory references to all memory spaces. Once the prediction is made, both user and kernel memory are maintained in the same predicted power state. If this state turns out to be PRE state, user memory occupied by the given task and the entire kernel memory will be fully powered during the task execution. This behavior may be detrimental to energy efficiency, if for example, the task is computationally intensive with low kernel activity. Subsequently, we extend the design of IAMEM by using two performance counters to monitor references to user and kernel memory individually. We further split MemoryRefs variable in Figure 5.3 into two fields UserRefs and KernelRefs. We note that user memory and kernel memory including the buffer cache are allocated into separate memory ranks to maximize management
Figure 5.5: The dual prediction algorithm.

efficiency.

The improved dual prediction algorithm is shown in Figure 5.5. Similarly to the previous algorithm, IAMEM first calculates the average task length $T_{avg}$, the average number of user memory references $M_u$ and kernel memory references $M_k$. Then based on the allowed task extension $E$, IAMEM calculates the lowest combination of power states for user memory $PS_u$ and for kernel memory $PS_k$, to keep the overall transition delays from both memory spaces below $E$. The calculation is first divided into two primary cases for testing which memory space will have more accesses. For each case, IAMEM tries from the lowest power combination to the highest power
combination, and the less accessed memory space will be given priority for a lower-power state over the more accessed memory space. This decision will consume less energy and incur lower transition delays than putting the more accessed memory space in the lower-power state.

5.1.5 Improving Monitoring Accuracy

So far, we have relied on monitoring memory accesses to estimate the worst-case transition overheads for a given task, by assuming that every memory access will require a power state transition. However, some of the memory references can arrive as a one cluster and only encounter one power state transition. To address this issue, we can use another performance counter to count the actual number of power state transitions that a given task encountered. This will allow IAMEM to update the variables of kernel and user memory reference in the prediction table with the actual number of memory references that encountered power state transitions. Subsequently, the algorithms in Figure 5.4 and Figure 5.5 remain unchanged except the $M$ variables now correspond to the numbers of power state transitions that occurred in user and kernel memory. Utilization of the actual power state transitions accounts for traffic burstiness and as a result, eliminates the inaccuracy resulting from monitoring only memory accesses in the original design.

We note that counting power state transitions is only available in low-power states (PD or SR) since PRE state does not require any power state transitions for memory I/Os. Not being able to monitor power state transitions for PRE state is insignificant since the deadlines of almost all tasks can be met with PD state, as shown by our experiments. Only extremely memory-intensive tasks will require memory in PRE state. Once such a task is detected, memory activity of the task will still be monitored using the previously described memory reference counters.
<table>
<thead>
<tr>
<th>Traced applications</th>
<th>Trace length</th>
<th>Num. of UI-IDs</th>
<th>Short tasks</th>
<th>Long tasks</th>
<th>Avg. task length</th>
<th>Avg. user memory ref.</th>
<th>Avg. kernel memory ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbiWord</td>
<td>3.1 hrs</td>
<td>179</td>
<td>22840</td>
<td>2080</td>
<td>0.02 sec</td>
<td>7794</td>
<td>2378</td>
</tr>
<tr>
<td>Gnumeric</td>
<td>2.9 hrs</td>
<td>342</td>
<td>7804</td>
<td>1416</td>
<td>0.05 sec</td>
<td>19979</td>
<td>5013</td>
</tr>
<tr>
<td>Anjuta</td>
<td>3.9 hrs</td>
<td>458</td>
<td>14300</td>
<td>1768</td>
<td>0.03 sec</td>
<td>4127</td>
<td>2457</td>
</tr>
<tr>
<td>GIMP</td>
<td>3.3 hrs</td>
<td>228</td>
<td>3356</td>
<td>2440</td>
<td>0.14 sec</td>
<td>39409</td>
<td>13660</td>
</tr>
<tr>
<td>LiVES</td>
<td>2.6 hrs</td>
<td>166</td>
<td>2728</td>
<td>1208</td>
<td>0.51 sec</td>
<td>67319</td>
<td>381907</td>
</tr>
<tr>
<td>Memtester</td>
<td>0.1 hrs</td>
<td>1</td>
<td>0</td>
<td>20</td>
<td>167.78 sec</td>
<td>264112456</td>
<td>10389729</td>
</tr>
</tbody>
</table>

Table 5.1: Statistics of application traces.

5.1.6 Preserving Performance

When a low-power state is predicted for a given task, the accumulated transition delays could become exposed to the user, resulting in performance degradation for longer tasks. To prevent excessive task extension, we propose an early-detect optimization to detect the possible user-perceivable degradation before the task completion. Observing that the scheduler in the kernel will interrupt task execution every 100ms to check if other processes should be scheduled, we add a monitoring module into the scheduler to monitor the given task execution for potentially missed deadlines. Every time the scheduler is invoked, the monitoring module reads the performance counter that counts the number of power state transitions from a low-power state (SR or PD) to PRE state and calculates the actual delay that the current running task has encountered so far. Once the delay exceeds the allowed 50ms extension, the monitoring module will raise the memory power state to the next higher level. If the next higher selection is PD state, the monitoring will continue. Seeing an additional delay of 50ms, due to the power state transitions, the memory power state is switched to PRE for the remaining task execution. This optimization will minimize the potential delays that are exposed to the user.

5.2 Methodology

We use trace-driven simulation to evaluate the proposed IAMEM and compare it with the following mechanisms:

- **PAVM.** Power-Aware Virtual Memory: The existing state-of-the-art per-
process mechanism which keeps the ranks occupied by the currently running process and the ranks occupied by kernel memory in Precharge state during the process execution, while keeping all other ranks in Self Refresh state.

- **ODPD.** On-Demand Powerdown: An existing mechanism that keeps all ranks in the system in Powerdown state during execution and makes transitions to Precharge state on memory request arrival. This is the special case of the Dynamic Rank Power Down technique implemented in Intel Core i CPUs, with the idle timeout value set to zero to minimize energy consumption.

- **ODSR.** On-Demand Self Refresh: We propose a complementary mechanism to ODPD that keeps all ranks in the system in Self Refresh state and transitions to Precharge state upon a memory request arrival.

- **ORACLE.** A per-task mechanism that utilizes the future knowledge to select optimal power states for ranks occupied by user and kernel memory for each incoming task.

Each of the evaluated mechanisms will put all ranks in the system to Self Refresh state when a task completes and the system begins idling. The simulator includes a task scheduler as well as a memory simulator. The memory simulator includes a memory controller and two DDR3-1066 DIMMs, each consisting of two 1GB ranks. The ranks are allocated to minimize fragmentation of memory for running processes across multiple ranks [24]. Finally, the energy management mechanism makes memory power decisions upon each UI event and the memory simulator executes the corresponding power-state transitions for the accessed ranks and calculates energy consumption according to Figure 2.2.

The application traces used in the simulation were collected using a modified Linux kernel 2.6.30 running on Intel Core i7-920 CPU with 4GB DDR3-1066. All traces contain data of UI events and process activity from a large number of usage sessions in the GNOME environment. UI events were collected with the modified X-Monitor, including the timestamp of each event and the interaction ID uniquely identifying the GUI component. Process activity traces were collected with Linux.
Trace Toolkit that logs program execution details from a patched Linux kernel. Based on the ordered event timestamps, we are able to simulate the dynamic execution progress of the traced applications, so that the computation time for each UI-triggered task can be calculated accurately.

The Core i7 CPU is equipped with a shared 8MB L3 cache; therefore, we set up two performance counters: one for counting the event MEM_LOAD RETIRED.L3_MISS that occurred in user mode, and the other for counting the same event in kernel mode [21]. Each of the counters was read as soon as an UI event was captured, and was read again upon the completion of the triggered task. The difference of the two counter values is considered as the number of references in user or kernel memory during this task execution.

We use five commonly executed interactive applications and one benchmark, shown in Table 5.1: AbiWord – a word processing application; Gnumeric – a spreadsheet application; Anjuta – an Integrated Development Environment for C/C++ and Java development; GIMP – an image processing application; LiVES – an integrated video editing and playback application; Memtester – a memory benchmark that intensively tests the performance of main memory. The traces were collected over a period of several hours and the trace length is shown in the second column. The number of interaction IDs presents the total number of unique user interactions in each application and serves as an indicator of the GUI complexity for the application. In case of Memtester, there is only one interaction to start the benchmark.

Table 5.1 also lists the numbers of short tasks (shorter than 50ms), long tasks (longer than 50ms), and the average task length for each application. Finally, the average numbers of memory references in user and kernel memory specifically indicate the per-task demand on the memory subsystem.

5.2.1 Performance Demand of Applications

Figure 5.6 shows the distribution of task processing demand for each application based on ORACLE’s optimal power selection for user and kernel memory that maximizes energy savings while eliminating delays exposed to the user. AbiWord, Gnu-
Figure 5.6: Task performance demand based on the ORACLE’s optimal power.

AbiW. and Anjuta have generally lower performance demand because most of their tasks require user think time to complete interactive operations such as editing text. Subsequently, user memory can stay in SR state for 53% of the time during the task execution. On the other hand, GIMP and LiVES work on large size images and video clips, resulting in significantly higher demand for memory performance. As a result, user memory has to spend 88% of the time in the higher performance states (PD and PRE) to prevent delays from being exposed. Finally, Memtester is a memory intensive benchmark that constantly reads and writes the memory and requires the maximum performance. Therefore, user memory must stay in PRE state all the time to prevent user perceived delays.

Figure 5.6 also shows that the performance requirements from kernel memory is lower than user memory. This is because the applications usually invoke few system calls and perform most processing in their own virtual address space, creating lower kernel memory activity as shown in Table 5.1. Subsequently, kernel memory can stay for 77% of the time in SR state for AbiWord, Gnumeric, Anjuta, and GIMP. However, LiVES and Memtester have higher kernel activity and the performance demand of kernel memory is comparable to user memory. As a result, kernel memory must also stay in the high power state for the majority of the time to prevent performance degradation.
5.3 Evaluation

5.3.1 Energy

Figure 5.7 shows the average per-task memory energy consumption for each mechanism and application. The energy bars are divided into four components: 1) energy consumed in SR state; 2) energy consumed in PD state; 3) energy consumed in PRE state; and 4) energy consumed for IO and power state transitions. The energy consumption of each mechanism accounts for all ranks in the system during the task execution, and is normalized to ORACLE.

PAVM always keeps user and kernel memory in PRE state during process execution, therefore consuming the most energy, 150% more than ORACLE. ODPD follows PAVM with 88% more energy consumption than ORACLE, because it keeps all memory in PD state during the task execution and it is clearly more than necessary for most tasks to eliminate user perceived delays, as shown in Figure 5.6. Additionally, ODPD does not attempt to optimize energy as other mechanisms, which only transition the accessed ranks to the appropriate state and keep others in SR state. IAMEM closely matches the energy consumption profile of ORACLE through sophisticated demand matching prediction, yielding close to optimal energy efficiency. Subsequently, IAMEM shows less than 3% difference in energy consumption from ORACLE, reducing the energy consumption of PAVM and ODPD by 53% and 37% respectively. In the best case occurring in AbiWord and Anjuta, where the lowest SR state can fit 60% of the time for user memory and 86% of the time for kernel memory, as shown in Figure 5.6, IAMEM yields as much as 70% improvement in energy efficiency as compared to PAVM and ODPD.

ODSR consumes the least amount of energy when we only consider main memory, yielding 26% less energy consumption than ORACLE, since all ranks are kept in SR state that has the lowest power demand. However, executing tasks with lower energy consumption than ORACLE is inefficient as it will expose delays to the user and may further increase the energy consumption of the entire system due to the longer runtime. Furthermore, ODSR misses the goal of this study for transparent
Figure 5.7: Average per-task memory energy consumption for processing each task normalized to ORACLE.

<table>
<thead>
<tr>
<th>Applications</th>
<th>ORACLE</th>
<th>PAVM</th>
<th>ODPD</th>
<th>ODSR</th>
<th>IAMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbiWord</td>
<td>3641.77</td>
<td>4189.16</td>
<td>3982.11</td>
<td>3612.91</td>
<td>3650.18</td>
</tr>
<tr>
<td>Gnumeric</td>
<td>3497.74</td>
<td>3953.39</td>
<td>3770.34</td>
<td>3445.60</td>
<td>3497.83</td>
</tr>
<tr>
<td>Anjuta</td>
<td>4525.16</td>
<td>5081.52</td>
<td>4878.22</td>
<td>4477.74</td>
<td>4522.34</td>
</tr>
<tr>
<td>GIMP</td>
<td>3981.66</td>
<td>4859.21</td>
<td>4508.02</td>
<td>3849.32</td>
<td>4018.42</td>
</tr>
<tr>
<td>LiVES</td>
<td>3978.27</td>
<td>5759.18</td>
<td>4877.01</td>
<td>3486.28</td>
<td>3993.54</td>
</tr>
<tr>
<td>Memtester</td>
<td>5529.43</td>
<td>5807.84</td>
<td>4396.73</td>
<td>2878.22</td>
<td>5672.02</td>
</tr>
</tbody>
</table>

Table 5.2: Total memory energy consumption (in Joules) during the entire application runtime.

energy optimizations that do not expose delays to the user. This scenario also occurs for ODPD in Memtester. As shown in Figure 5.6, Memtester requires PRE state for most of the execution time to avoid performance degradation. However, ODPD utilizes PD, a wrong power state, and consumes less energy than ORACLE. IAMEM still performs almost the same as ORACLE in this case, since it keeps using PRE state for user memory as required, while recognizing the relatively less demand for kernel memory performance and using the lower PD state when necessary.

While Figure 5.7 shows the energy consumption for processing tasks, it does not reflect the memory energy consumption over the entire execution time since the system idle time is not included. Each of the mechanisms puts all memory ranks in SR state when the system becomes idle, consuming the same amount of idle energy. Therefore, the overall improvements in energy efficiency originate from the energy
savings obtained during the task execution. Table 5.2 shows the total memory energy consumption during the whole program execution, including the idle time, for each application and each mechanism. As we can see, even considering the total execution time of several hours long (Table 5.1), IAMEM still gains significant energy savings over PAVM and ODPD, and matches energy consumption of ORACLE with less than 1% difference. For the first five applications, IAMEM consumes 17% and 11% less energy as compared to PAVM and ODPD, respectively. IAMEM energy reduction drops in Memtester due to the full power demand from the extremely memory-intensive tasks. Nevertheless, IAMEM still saves 4% energy as compared to PAVM in this case.

5.3.2 Performance

While reducing energy consumption is important, preservation of the performance is the goal of this research. Performance degradation can negate any energy savings and negatively affect the user experience. Figure 5.8 shows the average task length for each application and each mechanism normalized to ORACLE. The task runtime is divided into: 1) task processing time; and 2) the power-state transition time due to the transitions from the lower power state (SR or PD) to PRE state.

We notice that ORACLE introduces some amount of transition time into the task processing time as compared to PAVM that maintains the original task runtime. However, the shorter runtime in PAVM does not translate into better performance since users are not able to notice the shorter task completion and initiate any subsequent interactions. ORACLE always selects the best power state to fit the task execution within the user’s perception threshold. The included power-state transition time in ORACLE is not exposed to the user, keeping the user behavior unchanged just like in PAVM. Similarly, ODPD executes most tasks at the higher performance level than desired, resulting in excess energy consumption as shown in Figure 5.7. Therefore, a task that is executing longer than its execution in ORACLE exposes noticeable delay to the user, while running the task faster is not energy efficient.
Due to the large transition latency (957ns) from SR state, ODSR incurs the most performance degradation, prolonging task execution by 46% on average as compared to ORACLE. Memtester exposes the worst case for ODSR with 160% more delay exposed to the user. Memory intensive tasks in Memtester result in noticeable delays even in case of ODPD, which only encounters 7.5ns transition latency for each memory access. IAMEM dynamically recognizes memory intensive tasks and provides appropriate power state similarly to ORACLE, only exposing less than 1% delay to the user for each application. Combining the results from Figure 5.7 and Figure 5.8, we observe that IAMEM is the most energy efficient mechanism and almost perfectly matches the behavior of ORACLE. This indicates that utilization of the interaction context allows IAMEM to accurately predict the demand placed on the system and achieve near-optimal energy efficiency without performance degradation.

5.3.3 The Need for Dual Prediction

Figure 5.7 showed IAMEM with dual prediction for user memory and kernel memory separately, as described in Section 5.1.4. Alternatively, IAMEM may also view user and kernel memory as a whole, predicting only a single power state for both memory spaces. Figure 5.9 compares the average per-task energy of IAMEM with single power prediction (IAMEMC), normalized to ORACLE. The comparison is limited
Figure 5.9: Energy consumption of IAMEM with single and dual prediction.

to the ranks occupied by the running process (user memory) and the kernel (kernel memory), but does not include the other ranks in the system. This separation allows us to study the contribution of energy consumed by user and kernel memory to the total per-task energy consumption.

As shown in Figure 5.6, lower demand for kernel memory performance allows kernel memory to run in a low power state more frequently than user memory. Therefore, IAMEM with dual prediction significantly reduces the energy consumed by kernel memory, 23% less than IAMEMC, while the energy consumed by user memory stays relatively unchanged. Subsequently, IAMEM reduces the combined energy consumption of user and kernel memory by 11%, on average, as compared to IAMEMC. The significant benefit of the dual prediction justifies the need for predicting power individually for user and kernel memory.

5.3.4 Delay Reduction with Early-Detect

Preserving performance and bounding delays exposed to the user is critical for overall energy efficiency and the user’s satisfaction. Therefore, IAMEM adopts the early-detect optimization which uses the scheduler to check every 100ms as discussed in Section 5.1.6. This optimization will eliminate significant performance degradation for the long running tasks. Table 5.3 compares the average exposed delays of the
Table 5.3: Average delays of the delayed long running tasks in standard IAMEM with the early-detect optimization and alternative design without the optimization.

<table>
<thead>
<tr>
<th>Application</th>
<th>Num. of long running tasks</th>
<th>Num. of delayed long running</th>
<th>Delay w/ early-detect</th>
<th>Delay w/o early-detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbiWord</td>
<td>1315</td>
<td>237</td>
<td>21 ms</td>
<td>48 ms</td>
</tr>
<tr>
<td>Gnumeric</td>
<td>852</td>
<td>121</td>
<td>29 ms</td>
<td>110 ms</td>
</tr>
<tr>
<td>Anjuta</td>
<td>997</td>
<td>89</td>
<td>7 ms</td>
<td>32 ms</td>
</tr>
<tr>
<td>GIMP</td>
<td>1924</td>
<td>223</td>
<td>10 ms</td>
<td>60 ms</td>
</tr>
<tr>
<td>LiVES</td>
<td>857</td>
<td>71</td>
<td>23 ms</td>
<td>69 ms</td>
</tr>
<tr>
<td>Memtester</td>
<td>20</td>
<td>2</td>
<td>0.3 ms</td>
<td>212 ms</td>
</tr>
</tbody>
</table>

long running tasks (longer than the 100ms scheduler interval) for IAMEM with and without this optimization. The delays shown only count the extra times that exceed the allowed 50ms extension and may be noticed by the user. We can see that IAMEM with this optimization significantly reduces the amount of delays exposed to the user for long running tasks. The final small extension above 50ms is at the lower end of the perception threshold range (50-100ms), and will be unnoticeable to the user.

Alternatively, we can remove the optimization since IAMEM without early-detect still manages to keep the delays reasonable that may not be noticed by most users. In addition, IAMEM without early-detect would reduce the per-task energy consumption in Figure 5.7 by additional 2%, on average. However, the goal of this research is to maximize energy savings without affecting the user experience. The early-detect optimization is critical in achieving this goal, and we subsequently included it in IAMEM implementation and all previous results reflect the inclusion of this optimization. Furthermore, the impact of removing early-detect optimization on the overall energy efficiency, as shown Table 5.2, is negligible, reinforcing our decision to include it in IAMEM.
CHAPTER 6

CONCLUSIONS

Due to the increasing demand for large-scale data processing, memory has become one of the most energy-hungry components in the computer system. Data intensive applications require an efficient I/O subsystem and as a result the demand for a larger buffer cache is growing, surpassing the storage demand of virtual memory. Therefore, a low-power and high performance buffer cache is important for the overall system efficiency. To improve performance of power management mechanisms while providing high-energy savings, Chapter 3 presented several delay-hiding mechanisms for power state transitions of DRAM. We have shown that power state transitions can be efficiently hidden by standard I/O processing routines, and leverage that fact in proposing a range of mechanisms that efficiently hide transition delays. The best-proposed mechanism with early-turnoff energy optimization can hide almost all transition delays while only consuming 3% more energy than existing mechanisms.

In Chapter 4, we proposed and evaluated Interaction-Aware Dynamic Voltage Scaling (IADVS), a novel CPU voltage scaling mechanism that takes advantage of a human physical limitation, the perception threshold, to extend the runtime of interactive tasks by varying the CPU frequency without exposing new latencies to the user. We performed a detailed evaluation of a CPU managed by IADVS through simulation and verified our findings with an implementation and measurements on actual hardware. By relying on a transparent and robust fine-grained interaction capture system, IADVS improved CPU task demand prediction accuracy by 37% over the current state-of-the-art mechanism, reduced processing delays by 17%, and demonstrated a further 4% improvement in the CPU energy consumption. The analysis suggests that CPU energy management would benefit from per-core voltage settings in multi-core systems, as well as a wider range of hardware voltage settings.
for new CPUs.

As current applications are becoming more data-centric, computer systems are equipped with larger capacity and higher performance main memory. As a result, energy consumption of main memory is significantly increasing. In Chapter 5, we addressed interactive systems where most tasks are initiated by the user, and presented the design of IAMEM, a unified approach to manage the energy consumption of the entire memory space. By correlating the memory performance demand to the user interactions, IAMEM is able to accurately select suitable memory power states for UI-triggered tasks, saving energy while preserving the performance of the system. We have shown that compared to the state-of-the-art mechanisms, IAMEM saves 37%-53% of the memory energy consumed for task processing, resulting in up to 15% reduction of the total memory energy consumption during the entire program execution. In addition to the significant energy savings, IAMEM also successfully maintains the user-perceivable performance by hiding delays associated with energy management.
REFERENCES


REFERENCES – Continued


REFERENCES – Continued


REFERENCES – Continued


