UNIVERSITY OF CALIFORNIA
Santa Barbara

Modeling, Predicting and Reducing Energy Consumption in Resource Restricted Computers

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Selim Gürün

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Rich Wolski

Professor Tim Sherwood

March 2007

The Dissertation of
Selim Gürün is approved:

Professor Rich Wolski

Professor Tim Sherwood

Professor Chandra Krintz, Committee Chairperson

March 2007

Modeling, Predicting and Reducing Energy Consumption in Resource Restricted Computers

Copyright © 2007

by

Selim Gürün

To my parents, Mehmet and Fatma Gürün,

Acknowledgements

I dedicate this dissertation to numerous people who supported, trained and taught me during my entire professional and academic life. Without their support, I could have never achieved this stage of my career.

First off, I would like to thank my advisor, Prof. Chandra Krintz, for her relentless motivation, almost infinite patience and tremendous support. During the many years that she mentored me, she always let me explore a wide array of research objectives; at the same time, she also made sure that I had enough time for enjoying life in beautiful Santa Barbara. I will make every effort to become a professor or a mentor as good as Chandra. For everything she has done for me, I am and will always be grateful.

I would like to especially thank Professors Rich Wolski and Tim Sherwood for serving in my committee. In spite of their incredibly busy schedule, they not only found time to read my dissertation and gave precious feedback; they also helped me shape my future career goals.

I also would like to thank all my friends in Racelab and Mayhem. To Priya, for being a great climbing partner. To Hussam, for our loud, noisy backgammon games, and for finding the title of my dissertation. To Sunil, Ling Li, and Ye, for always being around, giving expert advices, and for helping me to focus on my research. To newer students in our lab, for their help in refining my dissertation ideas. I also want to thank

our department staff, especially to Mary Jane (now retired), Amanda and Bee Jay, for their incredible support.

I want to thank all my friends for making my life here much more enjoyable, and productive. Ahmet, Alphan, Aysu, Çağdaş, Emre, Erkan, Fatih, Işıl, Orhan, Özgür, Mustafa, Tolga and others. I will always miss the lunches and special dinners that we used to have. They created a community that is hard to repeat elsewhere.

Finally, my deepest thanks go to my family. To Fatoş, for her support and comforting during the ups and downs of a long Ph.D. education. To my parents, Fatma and Mehmet Gürün, for their lifetime dedication to education and well-being of their children. To my brother and sisters for their endless patiance. Without their love and support, none of this could ever be possible.

Curriculum Vitæ

Selim Gürün

Education

2006	Doctor of Philosphy in Computer Science, University of California, Santa Barbara, CA
1999	Master of Science in Computer Science, Rensselaer Polytechnic Institute, Troy, NY.
1997	Bachelor of Science in Computer Science, Middle East Technical University, Ankara, Turkey.

Experience

2002 – 2006	Graduate Research Assistant, University of California, Santa Barbara, CA
2000 – 2002	Software Engineer, Ericsson Inc, Santa Barbara, CA
1998 – 1999	Graduate Teaching Assistant, Rensselaer Polytechnic Institute, Troy, NY.

Publications

Selim Gurun, Chandra Krintz and Rich Wolski. *NWSLite: A Light-Weight Prediction Utility for Mobile Devices* (Extended version of Mobisys'04 work). Under submission in ACM Transactions on Embedded Systems

Selim Gurun and Chandra Krintz. *A Run-Time, Feedback-based Energy Estimation Model for Embedded Devices* (Extended version of Codes-ISSS'06 work). Under submission in ACM Transactions on Embedded Systems

Selim Gurun, Priya Nagpurkar and Ben Zhao. *Energy Consumption and Conservation in Mobile Peer-to-peer Systems*. In the proceedings of first International Workshop on Decentralized Resource Sharing in Mobile Computing and Networking (MobiShare), September, 2006

Selim Gurun and Chandra Krintz. *A Run-Time, Feedback-based Energy Estimation*

Model for Embedded Devices. In the proceedings of ACM International Conference on Hardware Software Codesign and System Synthesis (Codes-ISSS), October, 2006.

Ye Wen, Rich Wolski and Selim Gurun. *S2DB: A Novel Simulation Based Debugger for Sensor Network Applications*. In the proceedings of ACM International Conference on Embedded Software (EMSOFT), October, 2006.

Ye Wen, Selim Gurun, Navraj Chohan, Rich Wolski and Chandra Krintz. *Full System Cycle-Close Simulation of the Stargate Sensor Network Intermediate Node*. In the proceedings of IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS) July, 2006.

Chandra Krintz and Selim Gurun. *Remote Performance Monitoring*. In the proceedings of Schloss Dagstuhl Workshop on Automatic Performance Analysis, December, 2005.

Selim Gurun and Chandra Krintz. *AutoDVS: An Automatic, General-Purpose, Dynamic Clock Scheduling System for Hand-Held Devices*. In the proceedings of ACM International Conference on Embedded Software (EMSOFT), September, 2005.

Selim Gurun, Chandra Krintz and Rich Wolski. *NWSLite: A Light-Weight Prediction Utility for Mobile Devices*. In the proceedings of Mobile Systems, Applications and Services (MOBISYS), June, 2004.

Selim Gurun. *Addressing the Energy Crisis in Mobile Computing with Developing Power Aware Software*., UCSB Computer Science Technical report, 2003

(Book Chapter) Selim Gurun and Boleslaw Szymanski. *Automating Internet Routing Behavior Analysis Using Public WWW Traceroute Services*. Chapter in Managing QoS in Multimedia Networks and Services,J. Neuman de Souza and R. Boutaba, Springer, 2000

Selim Gurun and Boleslaw Szymanski. *Automating Internet Routing Behavior Analysis Using Public WWW Traceroute Services*. In Proceedings of the third International Conference on Management of Multimedia Networks and Services (MMSN), September, 2000.

Abstract

Modeling, Predicting and Reducing Energy Consumption in Resource Restricted Computers

Selim Gürün

Recently, mobile, battery-powered embedded devices such as personal digital assistants (PDAs), smartphones, and cellular devices, have become ubiquitous and increasingly capable. Worldwide, approximately 42 million smartphones and PDAs are shipped in the first half of 2006, and the predictions indicate that their sales will increase more than 57 percent by 2007. Given the proliferation and importance of these devices, users demand more capability from, and execution of increasingly complex applications on, these devices.

A key limitation on the utility of these devices is the battery. Since it is extremely difficult to increase battery supply, the best option for extending battery life is to use software techniques and systems that are power aware. The two most important techniques that reduce energy consumption are computation offloading and dynamic voltage scaling. In our work, we extend these techniques and investigate novel software solutions to enable power-awareness for real devices and real software.

The goal of both offloading and DVS systems is to extend battery life without impacting negatively the user's perception of program performance. Unfortunately, extant

approaches to both of these systems fall short in doing so. The primary reason for this is due to inaccuracies both in the measurement of past energy consumption and in the prediction of future program and workload behavior and resource availability. Thus, it is the goal of our work to devise novel techniques and infrastructures to improve the efficacy of these two power-aware optimizations.

In our work, we first develop techniques that measure energy consumption of tasks accurately. Our approach provides task energy estimations with a very low error margin (3.8% to 4.6%) Second, we present a set of prediction tools and strategies that make accurate forecasts of future application and resource behavior. Finally, we show how these techniques can be used to enable more effective offloading (27% to 56% less wasted energy when compared to its competitors, and DVS (31% to 49% savings of that has been previously possible). In all of our work, we consider real devices in use today and popular software systems and workloads.

Professor Chandra Krintz
Dissertation Committee Chair

Contents

Acknowledgements **v**

Curriculum Vitæ **vii**

Abstract **ix**

List of Figures **xiv**

List of Tables **xvi**

1 Introduction **1**

2 Background **7**
2.1 Computation Offloading 8
2.2 Dynamic Voltage Scaling 11
2.2.1 Interval Scheduling 14
2.2.2 Interactive Task Scheduling 15
2.3 Our Hardware And Software Setup 19
2.3.1 VPerfmon 23

3 Modeling Energy Consumption **26**
3.1 HPMs To Model Program Power Behavior 28
3.2 Modeling Methodology 30
3.3 Linear Regression For Modeling Computational Power Consumption . 34
3.3.1 Problem Encountered In Modeling 39
3.4 Modeling Communication Energy Consumption 46
3.5 Validation 49
3.5.1 Computation Model 50

3.5.2 Communication Model . . . 52
3.6 Why Linear Regression? . . . 54
3.7 Related Work . . . 55
3.8 Summary . . . 56

4 Predicting Energy Consumption at Run-Time 58
4.1 Extant OS Support For Measuring Energy Use . . . 59
4.2 Proposed Run-time Energy Prediction Mechanism . . . 61
4.3 Evaluation Methodology . . . 64
4.4 Results . . . 67
4.4.1 Decay Factor vs. Accuracy . . . 68
4.4.2 Update Period . . . 71
4.4.3 Benefits From Offline Profiling . . . 71
4.4.4 Battery Monitor Error Rate vs. Accuracy . . . 72
4.4.5 Performance Of Complex Model . . . 74
4.4.6 Performance Of Compact Model . . . 79
4.4.7 Execution Cost . . . 80
4.5 Why RLS-ED? . . . 81
4.6 Related Work . . . 83
4.7 Summary . . . 84

5 Predicting System Resources For Reducing Energy Consumption 86
5.1 Extant Resource Prediction Systems . . . 87
5.2 Proposed Non-parametric Resource Prediction Tool . . . 95
5.3 Design Rationale . . . 96
5.4 Validation . . . 105
5.4.1 Experimental methodology . . . 106
5.4.2 Evaluation Metrics . . . 108
5.4.3 Predictor Accuracy . . . 110
5.4.4 Computational Cost Of Prediction . . . 115
5.4.5 Validation Summary . . . 117
5.5 Summary . . . 121

6 Improving Computation Offloading 123
6.1 Resource Prediction in Remote Execution . . . 124
6.2 Methodology . . . 126
6.3 Simulation Results . . . 132
6.4 Summary . . . 139

7 Improving Dynamic Voltage Scaling **141**
7.1 Predicting User Interactivity For DVS 143
7.2 Design and Implementation 148
7.2.1 Monitoring GUI Events 151
7.2.2 CPU Load Sensor 153
7.2.3 Platform Specific Design Constraints 156
7.3 Collecting User Interactivity Traces 158
7.4 Evaluation Metrics 161
7.5 Results 163
7.5.1 Interactive Workloads 164
7.5.2 Concurrent Workloads 167
7.5.3 Integrating PACE 170
7.6 Summary 173

8 Conclusions **175**
8.1 Directions For Future Research 177

Bibliography **181**

List of Figures

2.1 Components of a typical remote execution system 9
2.2 PXA-270 power consumption for different voltage-frequency pairs . . 12
2.3 CPU performance scaling on a typical embedded processor 13
2.4 Test bed 19

3.1 Error rate for the computation model. 51
3.2 Error rate for the communication model. 53

4.1 Block diagram of proposed run-time power prediction system 62
4.2 Decay factor vs. accuracy 68
4.3 Static vs. adaptive models 70
4.4 RLS-ED update frequency vs. accuracy 70
4.5 Benefit from an offline profiler 72
4.6 RLS-ED execution cost 80
4.7 Recursive least squares memory shaping 82

5.1 Error matrix for a real input 102
5.2 Cost matrix for a real input 103
5.3 NWSLite relative error rate 113
5.4 NWSLite predictor predictability 114
5.5 Forecaster cost as number of instructions executed 116

6.1 Pseudocode for Scenario1 Decision Manager 126
6.2 Percentage of wrong decisions 134
6.3 Cost of wrong decisions 137

7.1 GUI event inter-arrival times for Solitaire 146
7.2 GUI event inter-arrival times for Tetrix 147
7.3 GUI event inter-arrival times for Opieplayer 147

7.4 AutoDVS policy stack and arbiter rules 149
7.5 iPAQ H3800 clock scaling request timing 152
7.6 Performance of AutoDVS and IDEAL for interactive workloads . . . 165
7.7 CPU utilization in Checkers 166
7.8 Performance of AutoDVS for interactive and soft-real time workloads 168
7.9 Simulated energy savings ratio with respect to AutoDVS 172

8.1 Power management using Foxton technology 178

List of Tables

3.1 Training and reference benchmarks 32
3.2 Coefficient and fit statistics for the computation model. 37
3.3 Correlation among model parameters 39
3.4 Principal components 43
3.5 Coefficient and fit statistics for improved models. 45
3.6 Communication energy model 48

4.1 Input variables in derived power models 65
4.2 Prediction benchmarks 67
4.3 Comparison of model error rates, updating every $\rho = 100$ intervals. . 75
4.4 Comparison of model error rates, updating every $\rho = 200$ intervals. . 76
4.5 Comparison of model error rates, updating every $\rho = 400$ intervals. . 77

5.1 NWS forecasters and the approximate costs of each 98
5.2 Datasets used for evaluation 105
5.3 NWSLite evaluation benchmarks 107
5.4 Error deviation for a set of representative traces 111
5.5 Execution cost comparison per prediction 116
5.6 NWSLite Results in summary 118

6.1 Power consumption of iPAQ under different scenarios [88] 130
6.2 Overview of 3-D objects. 132
6.3 Expected penalty for a wrong decision 138

7.1 Interactive events that we monitor 145
7.2 Intel SA1100 parameters 157
7.3 AutoDVS evaluation benchmarks and event traces 160
7.4 PPACE simulation parameters 171

Chapter 1

Introduction

Recently, mobile, battery-powered embedded devices such as personal digital assistants (PDAs), smartphones, and cellular devices, have become ubiquitous and increasingly capable. Worldwide, approximately 42 million smartphones and PDAs are shipped in the first half of 2006, and the predictions indicate that their sales will increase more than 57 percent by 2007 [83]. Concurrently with the proliferation and growing importance of these devices, users demand more capability from, and execution of increasingly complex applications on, these devices.

A key limitation on the utility of these devices is the battery. There are three ways to increase the battery lifetime in these devices: by increasing battery supply, by decreasing battery demand, or both [4, 14, 71, 18, 53, 59, 17, 16, 41, 88]. Unfortunately, it is very difficult to add to battery supply. The capacity of a battery depends on the chemical properties of the material that the battery is made of. New materials with more energy capacity are only made possible through complex, expensive and time consum-

ing research. The capacity of the most efficient battery has only increased 3 to 4 times in the last three decades despite significant effort [14]. Alternatively, it is possible to extend battery capacity by simply adding more batteries of the same type, however, this is highly undesirable since doing so increases the size, cost, and the weight of the device, which reduces devices mobility and cost-effectiveness.

To reduce the demand placed on the battery, we can reduce device hardware (capability) or use software techniques to optimize battery use. Reducing energy demand by cutting back hardware functionality is undesirable. The market trend and consumer interest continue to be towards devices that are more capable [82]. Newer and higher performance hardware components (which consume the battery at a faster rate), such as short and long range wireless interfaces, high capacity flash storages, and 32-bit CPUs are becoming increasingly common in mobile devices. The existence of these components is critical in many, key applications.

Thus, the best option for extending battery life is to use software techniques and systems that are *power aware*. Effective software approaches have become increasingly common in an effort to address this mobile computing energy crisis. In our work, we extend these techniques and investigate novel software solutions to enable power-awareness for real devices and real software.

In particular, this dissertation focuses on the infrastructure to support and enable the two most effective, extant, software optimizations for energy: Computation offloading

and dynamic voltage scaling [77, 28, 17, 18, 53, 26, 65, 41, 88, 64]. Computation offloading is a technique for executing parts of an application remotely on a more capable or wall-powered computer. This application uses reachable computer systems to extend the battery life and capability of resource-constrained mobile devices. The importance of this approach comes from the fact that offloading can reduce power consumption potentially with no performance degradation. Thus, it is very suitable for applications such as wearable computing, augmented reality, image processing, and speech recognition [53, 41].

However, the benefits of remote execution are highly dependent on numerous variables including computational complexity of the offloaded task, performance and interactivity expectations of the user, suitability of local and remote computation platforms, and network capacity. Offloading systems must measure past and predict future behavior, resource availability, and energy consumption for a wide range for resources: task execution and response time, network bandwidth, network latency, and CPU load and performance of the local and remote computer systems. If measurement or prediction is inaccurate, offloading systems can degrade performance significantly and consume additional energy.

Dynamic voltage scaling (DVS) is the process of changing the clock frequency and voltage of the mobile device during execution of programs and workloads [77, 28, 17, 26, 41, 88, 64]. DVS trades off performance for energy savings. By reducing

the voltage and the frequency of the CPU, we can have a quadratic effect on energy reduction. As a result, dynamic voltage scaling has the potential for reducing energy consumption significantly. However, to ensure that DVS is transparent to the user, we must use it in a way that best balances performance for energy savings. In particular, we must predict when the future CPU demand and unused CPU capacity enables us to reschedule the CPU clock without any perceived execution latency. As is the case for computation offloading, the prediction of these resources and program behaviors must be very accurate for DVS to be useful.

The goal of both offloading and DVS systems is to extend battery life without impacting negatively the user's perception of program performance. Unfortunately, extant approaches to both of these systems fall short in doing so. The primary reason for this is due to inaccuracies both in the measurement of past energy consumption and in the prediction of future program and workload behavior and resource availability. Thus, it is the goal of our work to devise novel techniques and infrastructures to improve the efficacy of these two power-aware optimizations. First, we identify and develop new techniques for accurate measurement of energy consumption. Surprisingly, the only mechanisms available on existing mobile devices for energy measurement are coarse-grain battery monitors that are highly inaccurate. With inaccurate measurements of past consumption behavior, it is virtually impossible to make accurate predictions of future behavior. Second, we present a set of prediction tools and strategies that can be used on

resource-constrained devices, with low overhead, that make accurate forecasts of future application and resource behavior, including task execution time, interactivity session length, wired and wireless network bandwidth and latency, and CPU load and availability. Accurate forecasts for these behaviors and resources are vital for the efficacy and thus, wide-spread use of computation offloading and DVS. Moreover, we provide these forecasts in a unified prediction framework that requires no input, calibration, offline execution of programs, or any other type of user participation. No such system exists that does so to our knowledge, prior to our work. Finally, we show how our approaches and energy-aware software systems can be used to enable more effective offloading and DVS that has been previously possible. In all of our work, we consider real devices in use today and popular software systems and workloads.

We organize the dissertation as follows. Chapter 2 presents the necessary background of our work and overviews existing systems on which we build and extend. This section describes two power saving strategies on which we focus: dynamic voltage scaling and computation offloading in greater detail. We also use this section to describe the characteristics of our empirical evaluation platform and to and present the energy metrics that we use. Chapter 3 discusses novel ways we can model energy consumption to enable accurate measurement of energy consumption on battery-powered, mobile and resource-constrained devices. Chapter 4 presents a run-time, dynamic energy estimation mechanism for these systems. Chapter 5 discusses prediction mecha-

nisms that exist in current systems and shows how they can be improved. Chapter 6 and 7 discuss how we extend battery life by using our measurement and prediction techniques for offloading and dynamic voltage scaling. Finally, Chapter 8 concludes the dissertation presents a summary of our key contributions.

Chapter 2

Background

While there are many strategies to reducing energy consumption of an application, two have proven to be the most effective in doing so: Computation offloading and dynamic voltage scaling (DVS) [77, 28, 17, 18, 53, 26, 65, 41, 88, 64]. In this dissertation, we present the infrastructure and support system to improve and enable these optimizations to achieve energy saving levels that are significantly higher that those that are available today. In this chapter, we overview each of these complementary optimizations and their related work to expose the infrastructure necessary to enable their efficacy in extending battery lifetime in battery-powered, resource-constrained devices. We also overview the target platforms on which we focus and detail the empirical evaluation setup that we use (Section 2.3).

2.1 Computation Offloading

Computation offloading (which is also known as remote execution in the literature) extends the computational power and battery life of battery-powered devices by partially executing tasks on more suitable computers [19, 65, 64, 41, 88]. Offloading systems attempt to reduce power consumption potentially with no performance degradation. Thus, it is very suitable for interactive tasks and multimedia applications with soft deadlines that impose implicit performance restrictions on hardware. Offloading has been shown to be effective for reducing demand on the local device and extending battery life for applications such as wearable computing, augmented reality, and speech recognition [53, 41, 18, 65, 64].

The benefits of remote execution are dependent on numerous variables which include the computational complexity of the offloaded task, the performance and interactivity expectations of the user, the suitability of the load on the local and remote computation platforms, and the network capacity. Consequently, if not executed appropriately, remote execution can lead to decreased performance and increased energy consumption.

Figure 2.1 depicts the general design of a remote execution system. A remote execution system offloads application tasks from battery-powered mobile devices to wall-powered, higher-performance servers. To decide whether a particular task should be

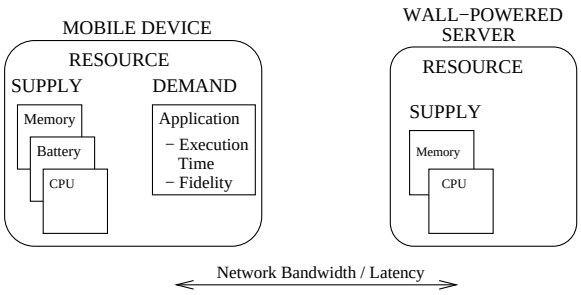


Figure 2.1: Components of a typical remote execution system. The decision process includes forecasting the available resource supply both at the client and server and application resource demand.

offloaded, a remote execution system must first compute the *resource demand* of the application task. Demand can be defined using different metrics such as CPU cycles, network bandwidth, memory pages, etc., according to the overall goals of the system.

To determine how best to accommodate demand, a remote execution system must evaluate how best to employ its *supply* – the set of resources, local and remote, that it has available to it for task execution. The system computes whether computation off-loading will be beneficial, according to its set of constraints, using a cost model. When cost of local execution exceeds that of remote execution, the system off-loads work to the server. The cost model must consider both the task execution characteristics as well as the highly-variable performance of the underlying resources that dictate computation and communication performance. However, constructing an exact cost function is non-trivial since hardware components have many shared resources, such as buses and DMA devices, that implement specific arbitration and priority policies.

While computation offloading is a powerful technique that can reduce computational requirements and power consumption significantly, its abuse can easily lead to counterproductive results. When the cost of network transfer is too high, or when the remote server is too loaded, the cost of offloading can exceed the cost of local execution. To prevent this, we have to identify and compute the energy cost of major offloading cost constituents; offloading to remote server, waiting for remote computation, and bringing the results back to local machine. We also have to compute the local energy consumption, so we can compare which path (offloading vs. local execution) is best.

Computing the local and remote cost requires predicting local and remote CPU availability and demand, network state (latency and bandwidth), and energy consumption of tasks. In Chapter 4, we discuss how to measure task energy consumption. In Chapter 5, first we describe how extant systems measure and predict resources that are important for computation offloading (Section 5.1), and next we suggest an adaptive, dynamic resource prediction technique for these resources (Section 5.2). Finally, in Chapter 6, we discuss how much extra power savings are possible as a result of extra accuracy that our prediction technique provides.

2.2 Dynamic Voltage Scaling

In modern, embedded-device CPUs, a significant portion of energy is dissipated in the form of dynamic power consumption [67, 51]. Dynamic power is a function of CPU voltage and frequency and is approximated by:

$$P \propto V^2 f \tag{2.1}$$

Thus, reducing the voltage level provides energy savings that are proportional to the square of the voltage reduction.

As the above equation indicates, it is possible to reduce power consumption of a general-purpose processor by reducing its clock speed. However, reducing clock speed alone does not conserve energy since any reduction in CPU performance are offset by a proportional increase in task execution time. Key to the dynamic clock scaling is the dependency between voltage and clock speed; the CPU voltage can be lowered in proportion to CPU clock frequency. This provides substantial savings –executing a task in a 0.75V setting instead of 1.5V one reduces CPU energy consumption by almost 75%. In practice, the savings are slightly less because of static leakage.

Figure 2.2 shows the power consumption of an Intel PXA-270 CPU [32] at different voltage/frequency settings. The x-axis shows CPU frequency. The y-axis shows CPU power consumption. The numbers in the plot area show corresponding CPU core voltage for each frequency level. The upper and lower boundaries of the gray area show the

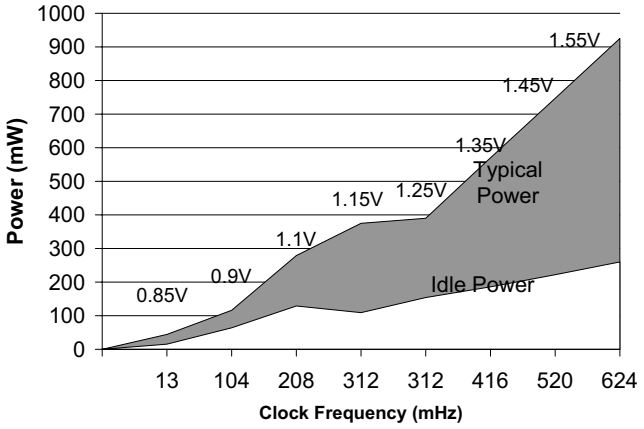


Figure 2.2: PXA-270 power consumption for different voltage-frequency pairs. The x and y axis show CPU frequency and power, respectively. The numbers in the plot area show corresponding CPU core voltage for each frequency level. The upper and lower boundaries of the gray area show the typical and idle energy consumption for the CPU, for each power/frequency pair. There are two 312 MHz setting, one with lower bus speed, and the other with a higher one.

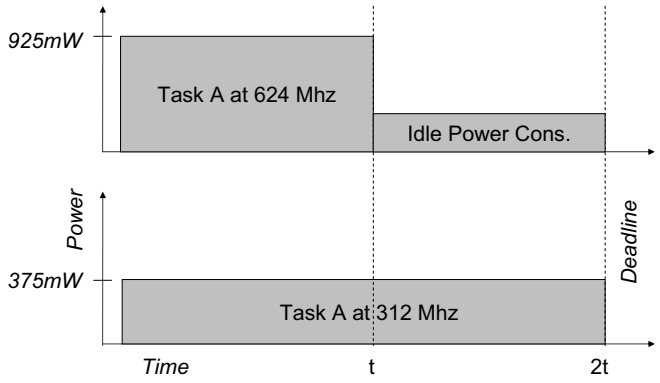


Figure 2.3: CPU performance scaling on a typical embedded processor. Executing Task A at full performance level causes it to finish well before the deadline, using more energy. Executing same task at lowered performance level uses less energy, and still completes the task before its deadline. We took the power numbers from PXA-270 manual [32].

typical and the idle energy consumption for each power/frequency pair. There are two 312 MHz settings, one with a lower bus speed, and the other with a higher one. For increasingly higher performance levels, the idle energy consumption is significantly less than typical active energy consumption level.

Scaling of the voltage requires scaling the frequency in the same proportion to meet signal propagation delay requirements [62]. Frequency scaling, and thus voltage scaling, results potentially in linear performance loss and reduced system responsiveness.

Figure 2.3 illustrates how dynamic voltage scaling attempts to reduce power consumption without imposing a perceivable performance loss on the application or workload. When the CPU has more CPU cycles available than a task demands, executing

the task at a lower performance level uses less energy, and completes the task before its deadline, while executing the task at full performance level uses more energy for executing the task and while idling.

To minimize perceivable negative performance impact and to enable energy conservation of voltage scaling, DVS policies must estimate future workload and choose the most appropriate CPU level. Accurately predicting future workload is challenging yet vital for efficacy and widespread use of DVS systems. Mis-prediction can result in setting the CPU level too high, curtailing power savings, or in setting the CPU level too low, producing an unresponsive system. There are two major existing approaches that address this challenge; interval schedulers and interactive task scheduling.

2.2.1 Interval Scheduling

Interval schedulers [77, 28, 26, 72, 3] divide the workload into fixed-length time intervals. These techniques use measurement history to estimate the workload in a future interval. For example, the PAST interval scheduler [77] assumes that the load in the next interval will be same as that in the last interval; the AVG_N interval scheduler [77, 28] assumes that the next interval is an exponential moving average (using a decay factor) of the N previous intervals. Other interval schedulers use observation heuristics [26] and more sophisticated statistical estimation methods [72] to estimate workload.

The efficacy of interval scheduling however, has proven to be limited in practice. Extant approaches use fixed-length, short intervals, i.e., 1-50ms, to accommodate for responsiveness requirements of interactive applications. However, for most applications the utilization pattern is visible only when deploying functions that span a larger period. For example, for an MPEG application, this pattern may not be visible even with a one second moving average [28].

Another limitation of prior approaches to interval scheduling is that they require a very short voltage switch latency (on the order of hundreds of microseconds) [28, 16]. Even though this rate is achievable in high-end specialized CPUs, e.g., the Transmeta Crusoe, most low-end handheld devices such as the HP iPAQ use much simpler hardware. Moreover the operating system must alert all synchronized peripheral devices that CPU speed is changing. These implementations can significantly increase the time required to complete a switch between frequency levels. Even though the improvements in CPU/hardware technology mitigate this problem a little bit, it may still be an important issue in practical implementations.

2.2.2 Interactive Task Scheduling

Prior DVS studies have focused on classifying tasks into different groups, each with a customized policy. [17] suggests three groups: Interactive, periodic, and background tasks. For interactive tasks, the system computes the optimum performance

factor (OPF). The OPF is the fraction of CPU speed required to complete a task no later than the user perception threshold. [17] defines this threshold as 50 milliseconds. The system in this prior work estimates the CPU load for a particular task in an interactive episode using an average of past CPU demand of the task, weighted by episode duration.

A periodic task consists of a producer-consumer pair. The system schedules a periodic task using an estimate of the time period between the completion of a producer and the start of a consumer. The system computes CPU speed such that producer ends immediately prior to when the consumer starts; the system uses the same CPU speed for both the producer and consumer.

Vertigo [16] is a refined and simplified implementation of this approach. Instead of categorizing tasks as producer and consumer, Vertigo maintains individual CPU utilization statistics for each task. It recomputes CPU utilization each time it attempts to reschedule a task. To identify interactive tasks, Vertigo monitors GUI events. When an event arrives, it marks the window manager and the recipient of GUI event as interactive. If any task communicates with an interactive task, Vertigo marks it also as interactive. The interactive period continues until all marked tasks are pre-empted by other tasks. Marking can be quite complex to implement, as tasks can use a variety of methods to communicate. Unfortunately, the implementation details and source code of Vertigo are not publicly available.

Lorch et al. suggest an approach that specifically targets user interactivity [49]. The system labels a user event with the type of GUI event that initiates it, e.g., a key-press, mouse-click, or drag event. Each event type has a separate DVS policy. The authors of this approach compute the CPU schedule using PACE [48]. PACE is a heuristic that the authors have proven to be optimal for computing CPU speed when (a) CPU can change frequency on a continuous scale, (b) all task deadlines are known, and (c) the cumulative distribution function (CDF) of task CPU demand is known.

All approaches that target user interactivity must overcome the challenge of determining when a task will complete without assistance from the application. The approach in [17] requires task completion time to update task execution time; the approach in [49] uses task completion time to compute task CDF and deadline. The former solution is precise, but is inherently complex; it requires monitoring system calls and communication between threads. The latter solution suggests an event is complete if a new event is posted or the idle thread is running and no I/O is ongoing. Even though this second approach can occasionally mis-classify a task as complete, it is more attractive due to its simplicity.

For the dynamic voltage scaling to be beneficial, we must make sure that, (1) energy cost of task execution at reduced performance level is lower than executing the task at higher performance level, (2) extra execution latency at lower performance level is transparent (or at least acceptable) to the user. While it is possible to compute (1) by

approximating the energy cost using Equation 2.1, achieving (2) requires measuring and predicting task demand, CPU availability and user expectations (tolerance to latency). In achieving (2), the techniques above use past resource measurements to predict the future, using exponential smoothing techniques. Chapter 5 discusses these prediction techniques. Here, we also discuss our own approach to predicting future state of CPU, both for demand and availability.

The interactive scheduling techniques do not try to predict an acceptable user latency (which is hard), instead, they assume a fixed deadline (user perception threshold of 50 milliseconds [69]) for each user interactive task and schedule the tasks such that they execute within the deadline. However, task demand has to be known. They do this either using cumulative distribution function, or profiling, or at run-time, dynamically. Furthermore, they have to measure and predict CPU availability to compute task execution latency. Section 5.1 of Chapter 5 discusses these in great detail.

Interactive tasks are not the only type of tasks in an interactive system; there are batch tasks that are executed either in association with the interactive ones, or as stand alone (MP3 decoding, OS daemons, etc). An interactive DVS system has to combine both to successfully schedule task execution in a power effective way. In Chapter 7, we discuss the design, and validation of such a system in a popular resource-restricted computer.

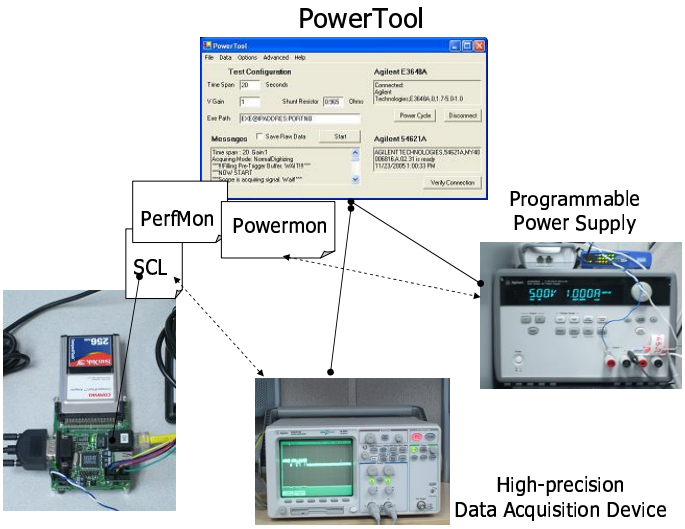


Figure 2.4: Test bed

2.3 Our Hardware And Software Setup

Accurately understanding and characterizing energy behavior is critical for the techniques that are developed and evaluated in this dissertation. We do this using a state-of-the-art test bed. The test bed includes a high-precision oscilloscope, a variety of embedded computers, a power supply and a desktop computer that collects and analyzes data.

Figure 2.4 shows the the test and measurement setup. It includes a tightly integrated suite of four tools to monitor program energy, power, and CPU performance.

- A device driver and Linux kernel patches, called VPerfmon, that enable and control HPM, power, and energy profiling.
- A user program, called VPMon, that executes a submitted program under the control of VPerfmon.
- A user program, called SCL, that dynamically switches CPU frequency level.
- A Windows XP GUI program called the PowerTool, that monitors and controls the lab equipment (oscilloscope and power supply), and sets the experimental parameters.

The test setup monitors program power consumption at a very fine granularity (thousands of times/second) and high accuracy (1mW resolution) using the data acquisition device. The setup can measure the energy consumption of the device and its PCMCIA cards (the wireless card and the compact flash) independently, and concurrently.

VPMon executes on a Linux-operated *target device*. It acts as the user interface to the target device that executes the submitted program and controls and collects hardware performance monitors (HPM) data by interacting with VPerfmon. VPMon and VPerfmon are portable to any platform that runs Linux and implements hardware performance counters.

Our target device is the Stargate sensor network intermediate node. The Stargate is representative of modern battery-powered, resource constrained devices as it implements the recent PXA-255 XScale processor and a wide range of popular I/O devices. Architecturally, and it terms of computational resources, it is similar to an HP iPAQ H5550 without any LCD display. Both devices have 400 MHz CPU, 64MB RAM, and can operate under same Linux operating system. The Stargate however, has its power supply pins exposed. Thus, it is more suitable for measuring computation and communication power consumption.

The test setup consists of an Agilent N54621A deep-memory oscilloscope that monitors the current passing through a high-precision resistor connected to the Stargate power supply. The oscilloscope has a connection to a workstation through a general purpose interface bus (GPIB). The PowerTool executes on the workstation and reads, analyzes, and saves the collected data. The PowerTool also controls a high-precision, programmable power supply, the Agilent E3648A. In power measurement setups that are powered by battery or wall power, fluctuations in supply voltage introduce an additional error. The Agilent E3648A minimizes these errors by providing a precise, constant voltage supply to the embedded test board.

The oscilloscope samples the voltage drop on the resistor 10000 times per second and records the measurements in its memory in real time. We create the waveform at offline by interpolating measurements after we download them from the oscilloscope

to our local machine. We use the interpolated data to measure energy consumption at a particular time. Mapping this information to operating system/software threads requires knowing when these tasks were executing –in oscilloscope time (since the energy samples were collected using oscilloscope timestamps). Unfortunately, the oscilloscope and our target embedded board operate their own, independent clock domains, thus this information is not readily available. We solve this problem by synchronizing oscilloscope clock. By connecting one oscilloscope channel to an output port of the embedded computer, we signal the enter/exit times of operating system threads to the oscilloscope. We then use this information to identify when a task entered and exited, and how much energy it consumed. Again, this computation is done at offline, with no overhead to run-time system.

The SCL driver scales the CPU speed of the Stargate if desired. The Stargate processor, the PXA-255, has a very flexible CPU clock implementation that users can configure to set memory, bus, and CPU core speed independently. There are currently five valid configurations (due to timing constraints). SCL enables users to manipulate the configurations at runtime and compiles a log of the new speed, device, and the time at which it implemented the changes the clock speed (using microsecond resolution).

2.3.1 VPerfmon

VPerfmon is the control center for program profiling. VPerfmon provides virtual hardware performance counters to each application. The HPMs by default count global CPU events, i.e. they do not track events at the program or thread level. VPerfmon provides a layer that multiplexes the counters and that enables selective monitoring of particular programs and threads. VPerfmon implements a virtual instruction per cycle (IPC) counter by tracking instructions (cycles are tracked by default on most devices). The virtual counters are 64bits in size to reduce overflow problems. It is possible to selectively enable/disable sampling during the monitoring. This functionality is similar to the middlewares like PAPI [58]. However, the extant middleware does not support our target CPU at present.

In our target device, the Stargate processor, the PXA-255, implements three 32-bit event counters; the hardware uses one to monitor dynamic clock cycles. VPerfmon sets the remaining counters to any two of the 14 events supported. The VPerfmon virtual counters reflect the same architecture ((i.e. extended to 64 bits), it uses one counter to count CPU clock cycles and the other two to monitor events.

VPerfmon interfaces to and monitors other system events to increase the accuracy of the HPM profiles. When the VPMon initiates a new program, it contacts VPerfmon. The VPerfmon driver allocates a set of virtual counters for the new task. Similarly, VPerfmon allocates a set of virtual counter when a process under the control of VPerf-

mon forks a child process. When the kernel performs a context switch to a task under VPerfmon control, VPerfmon configures and enables the counters. When the task is suspended or terminates, VPerfmon stores the virtual HPMs.

To isolate application and operating system performance, the VPerfmon kernel patch disables HPMs on interrupt entry and re-enables them on exit. This operation requires a read-modify-write cycle that is equal to three XScale instructions. As a result, the patch does not significantly increase interrupt latency.

VPerfmon facilitates interval-based data collection via the GPIO pin on the development board. Initially the GPIO pin is reset to logic 0 on program start. During program execution, VPerfmon toggles the pin's value at then end of every interval. VPerfmon, as mentioned above tracks interval lengths (arbitrary or fixed) using some performance event specified by the user. For the data in this paper, we use instruction counts as the event and fixed-length intervals of 10 million instructions. The oscilloscope is equipped with two channels. One channel monitors the voltage shunt resistor to measure power consumption. The second channel monitors the GPIO pin that VPerfmon toggles. Using this setup, our setup is able to log and track power, energy, and performance data at interval boundaries.

We use the described setup to measure task energy consumption, to develop our power models and to validate our results. In Chapter 3 and Chapter 4, we discuss energy modeling, measurement and prediction at run-time. Developing and validating

these models require collecting an enormous amount of data, including thousands of samples over a large number of benchmarks (given in respective chapters). Having such a setup reduced the overhead of data collection time significantly while improving the accuracy of the collected data.

Chapter 3

Modeling Energy Consumption

Understanding and accurately characterizing energy consumption is key to optimizing energy consumption of resource-restricted devices. Energy consumption of a program is highly variable depending on executed instruction type, CPU performance level, memory and I/O activity. In order to optimize power behavior effectively, power aware systems have to know how much a program task costs in terms of energy. While it is possible to define this cost using program execution time, CPU cycles, or similar metrics, these metrics cannot capture the variation in program power consumption. Power must be measured using its own metric, Watts and Joules.

There are extant tools that measure program energy consumption accurately. However, these tools require highly specialized setups. Powerscope [21] and JouleTrack [73] are two of these. Using very precise lab equipment, and power simulators, they create profiles that attribute program energy consumption to individual tasks and threads. Developers can use this information to identify and optimize the tasks that consume sig-

nificant amounts of energy. The progenitors of these tools have shown that significant power savings are possible from optimizing tasks during program development.

Without similar tools however, we cannot accurately measure task power consumption on the device itself while it executes applications. This is because, current device technology exports only inaccurate, coarse-grained battery level information. This data can only be detected at large measurement intervals (precluding attributing energy consumption to fine-grained program activities such as instructions). Moreover, this data fluctuates, is non-monotonic, and inaccurate due to environmental and chemical effects and to efforts to keep the cost of batteries low. Unfortunately, accurate, online measurement is key to enabling dynamic optimization techniques that extending battery life. Such techniques use dynamic profile information of the battery consumption of a task or program to estimate future energy consumption and to identify opportunities for optimization [87, 65, 55, 86, 47]. If the measurement data of task activity is coarse-grained and inaccurate, these techniques will make incorrect decisions that limit their energy savings or actually cause the system consume more energy than it saves.

The goal of this chapter is to analyze and understand the difficulties in modeling full system energy consumption. While its main goal is to pave the way to a dynamic, run-time power prediction system (which we present in next chapter), this chapter limits itself to exploring challenges in a static, offline model. This chapter starts with a discussion of use of hardware performance monitors in modeling energy consumption.

It proposes that these counters should be complemented with software counters for high level energy estimation. Section 3.2 presents modeling methodology. Section 3.3 discusses modeling computational energy consumption and presents a way to resolve the dependency related problems within the linear model. Section 3.4 discusses how to model I/O devices that do not have hardware performance monitors. Section 3.5 evaluates the proposed ideas by comparing model output to measurements collected using a real device. Section 3.6 discusses other modeling approaches. Section 3.7 gives related work. Finally, Section 3.8 discusses our findings and concludes the chapter.

3.1 HPMs To Model Program Power Behavior

Most modern processors have a hardware performance monitoring unit (HPM) that capture CPU performance data and make it available to developers and users. The monitoring unit has a set of accumulator registers. When enabled, these registers count the occurrence and duration of major hardware events, such as TLB misses, cache hits, branch mispredictions, etc.

HPM events provide significant insight into program behavior –indeed, these events have been used successfully to model CPU energy consumption in many studies [7, 12, 36, 35, 39]. These studies develop models for energy consumption of the CPU alone and CPU and memory subsystem in isolation. Their findings show that first-order,

linear models that map HPM events into CPU power consumption can achieve high accuracy. In our study, we use HPM counters to model full system power consumption.

Modeling full-system energy consumption is more complex than modeling CPU energy consumption. First of all, it is hard to monitor memory activity accurately [12, 39]. The challenge in memory is that, at present, HPM events provide little insight into memory behavior –there is no direct HPM event that monitor memory access rate in most CPUs. In addition there are no hardware events that provide insight into I/O activity. Many I/O devices have their own firmware or microcode that can asynchronously change their power states. These state changes are completely transparent to CPU and the HPM unit. Furthermore, large I/O data transactions generally use direct memory access (DMA), –the HPMs cannot capture these.

Even though these micro level transactions cannot be captured by HPM counters, [46] shows that macro level changes (i.e. a lower instructions per clock cycle, higher data stalls, etc.) in program behavior are enough to model I/O power consumption. The study in [46] is developed using a device with a SCSI disk, 3-level memory hierarchy and a MIPS CPU. It uses a static, customized linear regression model for each individual operating system routine, and demonstrates a high prediction accuracy.

Unfortunately, modeling each I/O call individually is not trivial. There are numerous I/O devices and same I/O call (such as read, write) can be used to access a number

of different devices. Modeling and maintaining a power model for each I/O device and an operating system call is not scalable.

This chapter proposes a different approach to monitor I/O energy consumption. Here, our approach is to capture I/O behavior with a carefully chosen combination of software and hardware performance counters and combine it with a carefully chosen set of HPM counters to predict full system energy consumption. The rest of this chapter substantiates this approach. The next section develops a computational model using only hardware counters. It only models energy consumption of core components; CPU and memory. Next, Section 3.4 develops a communication energy consumption model to evaluate the idea of monitoring I/O energy consumption.

3.2 Modeling Methodology

To explore the relationship between program behavior and power consumption, We use the Stargate platform instead of the iPAQ handhelds. Even though, both platforms are similar in their hardware and software capabilities, Stargate has many advantages over iPAQ for such a study. In Stargate, the power supply pins of both the main board and PCMCIA cards are easily accessible without breaking apart the device, which is not the case for iPAQs. Furthermore, the Stargate has multiple I/O ports that become quite handy in designing a power measurement setup.

The experimental setup includes multiple Stargate sensor network gateways and H5550 iPAQs running Linux 2.14.19, an Agilent 54621A oscilloscope and an Agilent E3648A variable power supply. The oscilloscope profiles energy consumption of one of the Stargates. We monitor energy consumption in fixed length intervals of 10 million instructions. A device driver on the Stargate configures the hardware performance counters, (i.e. HPM), to generate an interrupt after each interval. The interrupt handler collects HPM data and forces a logic transition on an output port. The Agilent oscilloscope records these transition times and voltage/current data at a rate of 10000 samples/second. Offline, we analyze this data to extract the length of each interval, peak and average power consumption, and total energy consumption.

To validate the idea of using software counters for modeling communication device, we profile the energy consumption of a Netgear 802.11b during wireless communication. In the setup, there is a Netgear 802.11b network card on each Stargate; the iPAQs have their own internal 802.11b cards. We configure all the hosts to the 11MB/s ad-hoc mode, in direct line of sight of each other.

We construct two models to evaluate the counter-based profiling idea. The first one profiles the energy consumption of software when I/O is not a concern. The second one profiles I/O heavy applications. When there is no I/O present, the software counters do not provide any additional information. Therefore the first model uses only HPM

Training Benchmark Set		Reference Benchmark Set	
Application	Description	Application	Description
basicmath	Math Test	gsmdecode	GSM decoder
dijkstra	Dijkstra shortest path	gsmencode	GSM encoder
matmult	Matrix multiplication	jpegdecode	JPEG decoder
stringsearch	String search	jpegencode	JPEG encoder
memri*	Memory read in-cache	mpegdecode	MPEG decoder
memro*	Memory read out-of-cache	mpegencode	MPEG encoder
memwi*	Memory write in-cache	em3d (Java)	Graph processing
memwo*	Memory write out-of-cache	bisort (Java)	Sorting
reg*	Register operations	treeadd (Java)	Recursive depth-first traversal
scps	secure file send	scps	secure file send
scpr	secure file receive	scpr	secure file receive
netpipe	network analyzer	game of life	MPI life game
		pvnx	MPI solver (small)
		pvkx	MPI solver (medium)
		pvkxb	MPI solver (large)

Table 3.1: Training and reference benchmarks. Training benchmarks (left) parameterize the model and Reference (right) benchmarks evaluate the accuracy of it. The benchmarks above the line are to model/evaluate computation; those below for communication. The applications with asterisks are home-grown.

counters to predict computational energy. The second one uses both. We call the former *the computation model* and the latter *the communication model*.

To develop these models, we use a large set of applications. The first suite, to which we refer to as the training set, we use to define our model. The second suite, to which we refer to as the reference set, we use for the empirical evaluation of the accuracy of our model. We present the suites and their brief description in Table 3.1. The left half of the table is the training set and the right is the reference set. We use the benchmarks

above the line to model/evaluate computation and those below for communication. We execute all programs from RAM drive to minimize the effect of flash read/write latency. The wireless network card is on for all experiments regardless of whether it is used or not.

The applications come from popular benchmark suites (e.g. MediaBench [45], Mibench [29], and Java-Olden [5]). The communication benchmarks include the secure copy protocol (scp) receive and transmit and netpipe [63]. For scp, we transfer a 1.7 MB file. Netpipe is a network analyzer. We also include distributed (message passing interface (MPI)) applications: Game of life [24], pvnx, pvkx and pvkxb [75]. MPI is typically employed for distributed computing applications in larger systems. These MPI applications have moderate computation requirements that are within the limits of the Stargate.

The characteristics of the MPI applications are analogous to the requirements of high-performance sensor network applications. For example in Life, the first processor divides the problem space into subspaces and distributes them to the other processors. Once the other processors complete the execution, they return the results back to the first processor. Then the first processor combines the results, and reiterates the process if necessary. This mechanism is very similar to recent query processing and vehicle tracking architectures for sensor networks. For instance in [68], the nodes are organized in a tree structure. The root node distributes a query to the network. Each node partially

processes the query and returns the results to the parent node. It is the parent node which combines the results. In [76], the remote sensor nodes collaborate with a central sensor node for airport security and tracking moving objects. The remote nodes do partial stream processing and filtering using computationally expensive algorithms, while they continuously exchange updates with a central node. The central node produces the results.

A device driver collects profile information of these applications during their execution. We develop and validate our proposed power profiling model using this data. The next section describes the model in detail.

3.3 Linear Regression For Modeling Computational Power Consumption

A linear regression equation models the relation between an output (response) variable y and input (explanatory) variables $x_1, x_2, ..., x_k$ using:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_k x_k \tag{3.1}$$

Here β_i are model coefficients. Assume we observe the physical relation between y and x_i, n times:

y	x_1	x_2	...	x_k
y_1	x_{11}	x_{21}	...	x_{k1}
y_2	x_{12}	x_{22}	...	x_{k2}
.	.	.		.
y_n	x_{1n}	x_{2n}	...	x_{kn}

We can rewrite this relation using the matrix form $y = X\beta + \varepsilon$ and solve it using $b = (X'X)^{-1}X'y$ [52]. Here, b is the least squares (LSQ) estimator for linear model coefficients β. In our model, x_i are hardware and software event counts that are highly influential in energy consumption and y_i are the energy measurements (in Joules). We experiment with different model sizes k. We collect event counts and energy measurements by sampling program execution every 10 million instructions. LSQ models are simple and robust and they do not require a priori knowledge of the distribution associated with the observations [23].

Our proposed energy estimation model consists of a computation and a communication component. This chapter only describes the computation model. The communication model is described in Section 3.4. The computation model estimates the energy consumption on average per instruction for tasks that do not have any significant persistent storage access or communication behavior. In other words, the computation model models the energy consumption of three most important unit; CPU, memory, and the memory bus.

To understand the relation between hardware events and the power consumption, we designed a large model that contains all major power related events. This model

employs 6 parameters: cycles per instruction (CPI) (x_1), instruction cache misses (x_2), instructions not delivered (x_3), data stalls (x_4), instruction TLB misses (x_5) and data TLB misses (x_6). These and similar events has been shown to be effective for power estimation of the CPU and memory [12, 36, 34]. The power model that we use is as follows;

$$E(nanojoules) = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_6 x_6 \tag{3.2}$$

where x's are the model input and the α's are the weights determined by the model. The model inputs are expected number of events for an average instruction, for instance x_2 gives the expected number of cache misses during the execution of an instruction. We compute this value by dividing the number of event counts in an interval by the interval length (10 million). The model outputs the estimated energy consumption of an instruction on average, in nanojoules. We estimate the parameter weights using least squares linear regression (i.e. LSQ) as we described above.

Since the XScale processor is only able to monitor two events at once, we execute the same program many times to monitor the different events. The measurement data can differ across runs (of the same program/input) as a result of hardware state or operating system events. To be able to better understand the extent of such perturbations, we monitor each event 5 times. However, *since averaging causes linear regression to look stronger than it really is* [23], We only use the 3rd dataset for each event. We

Computation Energy Consumption Model				
	Description	Coefficient	T-stat	P-Value
α_0	Constant	-0.19	-0.73	0.46
α_1	CPI	7.06	38.78	0.00
α_2	Inst. Miss	678.07	0.55	0.58
α_3	Inst. Not Dlvrd	-4.28	-1.61	0.11
α_4	Data Stalls	-1.07	-5.99	0.00
α_5	Inst. TLB Miss	686.06	-0.02	0.98
α_6	Data TLB Miss	-593.39	-8.33	0.00
R^2		0.99		
Average Error		3.80%		

Table 3.2: Coefficient and fit statistics for the computation model.

use the remaining 4 observations to evaluate the impact of such perturbations on model accuracy.

The benchmarks that we use are significantly different in their durations. Since we do not want any single benchmark to be represented more than its share in our model, we choose an equal number of observations from each benchmark. To extend the range of possible behaviors, we select the first, middle, and last 10 intervals from each profile.

Table 3.2 presents the coefficients of model and evaluate its fit on the training benchmark. The top portion of the table shows the coefficients for each of the HPMs. The bottom portion of the table shows the fit statistics. We evaluate the accuracy of computation model for the reference set in Section 3.5.

The coefficient of determination, i.e., the R^2 fit statistic, indicates the amount of variation that the model explains. Under most circumstances, it is a reliable indicator

of model goodness. The R^2 varies between 0 and 1, and larger values are better. The high R^2 value of presented model is a positive indicator of its high quality.

The average error statistic shows the absolute model estimation error. We compute this value using $1/n \times \sum (|\text{measured} - \text{estimated}|/\text{measured}) \times 100$, where n is the number of measurements. The model fits very well to the data with an average error of 3.8%. The rightmost two columns show the statistical significance for the model coefficients. The t-statistic values show whether we can reject the null hypothesis that the coefficient of the parameter is zero. The larger values indicate that we have a better chance of rejecting the hypothesis. The final column shows the probability of having null hypothesis true (not reject) (i.e. the coefficient has a high probability that it does not influence the model output variable). It is equal to $Pr(|t| > t - stat)$, where t is a student's t-distributed random variable with (n-k) degrees of freedom. Here, n is the number of observations k is the number of regressors. In our experiment, n is equal to 260, and k is 6, since there are 6 regressors including the intercept. The t-statistic values indicate that only three of the coefficients in our model, CPI, DSTALL, and DTLBMISS are statistically significant (have a high probability of being different than 0).

There are two problems with the presented power consumption model. The minor one is that the model requires more input parameters than what Intel XScale can monitor at a given time. However since more sophisticated processors, like Pentium IV, can

	CPI	IMISS	INDLVR	DSTALL	ITLBMISS	DTLBMISS
CPI	1.00					
IMISS	-0.04	1.00				
INDLVR	-0.14	**0.89**	1.00			
DSTALL	**0.71**	-0.05	-0.15	1.00		
ITLBMISS	-0.04	**0.97**	**0.84**	-0.04	1.00	
DTLBMISS	**0.74**	-0.03	-0.11	0.05	-0.03	1.00

Table 3.3: Correlation among model parameters. The darker entries show events that have strong correlations.

easily support monitoring a dozen or more events at a time, it is highly possible that future embedded processors will have the same functionality. The second problem is more serious. Most of the coefficients that the model generates are negative –which is perfectly fine from a regression point of view, but highly disturbing since energy consumption cannot be negative. To understand the root reason, we investigate the model, its parameters and their dependency relations. The next section details this.

3.3.1 Problem Encountered In Modeling

Our approach to negative coefficients problem is two-fold. First, we use statistically valid ways to reduce the number of parameters in power consumption model. Here, the goal is to develop a model that is accurate but that employs a small number of parameters. A smaller model is more attractive since it can easily expose the complex relationship between parameters and let me validate the model.

To eliminate model parameters, we first analyze the statistical correlations between the events. Table 3.3 shows the correlation data. A correlation coefficient that is close to 1.0 means that event pairs are highly correlated.

The correlation matrix shows that CPI, data stalls (DSTALL) and data TLB misses (DTLBMISS) have high correlation. Among those three, we retain the CPI, since it has the highest t-statistic, and discard the others. Furthermore, the instruction cache (IMISS, INDLVRD) and instruction TLB miss (ITLBMISS) events also have strong correlation. Using similar reasoning, we retain IMISS and discard ITLBMISS and INDLVRD events. Next, we form a model from IMISS and CPI data that we refer to as *MMISS*. Furthermore, we also form a model, *MCPI*, which uses the CPI metric alone.

Our second method focuses on enabling the extraction of meaningful information from the coefficients of the model. That is, we would like to be able to understand the contribution of each component on model output. By doing so it becomes possible to estimate energy consumption of each hardware event -and it may be easier to understand why some are negative.

An interesting phenomenon in HPM data is the existence of *multicollinearity*. Multicollinearity indicates that some linear relationship exists between the model parameters. For example, data cache misses are related to data stalls. As the amount of linearity increases between metrics, the stability of the coefficient estimates decreases

precluding us from extracting useful information from the coefficients in computation model [23].

If the purpose of regression is only to estimate a variable y using a set of model inputs x_i, without assigning any particular meaning to the values of x_i, multicollinearity is not a significant problem and model predictions will still be accurate. However, if the goal is to understand how much each x_i effects y, then multicollinearity can lead to misleading results. Models that suffer from multicollinearity have much larger confidence intervals in parameter estimations, that is, the parameter estimations can shift substantially when there are small changes in input. Another side effect of such wide confidence intervals is that the t-statistic value of individual parameters cannot be used confidently to remove arbitrary parameters from the model.

Our approach for reducing multicollinearity is to apply principal component analysis [37] to transform the correlated variables into a smaller number of uncorrelated variables called as principal components. The first principal component captures as much of the variability in the data as possible, and the succeeding components capture the rest of the variability.

As a first step to PCA, we standardize the dataset, that is, we subtract sample mean from each observation, and then divide the result by standard deviation of each parameter.

Let the variables x_{ij} are model inputs,(i.e the explanatory variables), such that x_{ij} is equal to j^{th} observation of i_{th} variable. The variable $\bar{x_i}$ gives the observed mean of variable x_i that is $\bar{x_i} = \sum_1^n x_{ij}/n$, and s_i is the sample standard deviation of x_i that is $s_i = \sqrt{\sum_1^n (x_{ij} - \bar{x_i})^2/(n-1)}$. In both formulas, n is the number of observations.

After standardization, the goal is to find the coefficients γ_i that best fit to the linear equation:

$$y_j = \gamma_0 + \gamma_1 x_{1j}^s + \gamma_2 x_{2j}^s + \gamma_3 x_{3j}^s + ... + \gamma_6 x_{6j}^s + \varepsilon_i \tag{3.3}$$

In the equation above, x_{ij}^s are the standardized observations of HPM counters. y_j is called the response variable or model output and here it gives the energy consumption per instruction. The ε is the error term (which we assume to be normally distributed). One can rewrite Equation 3.3 as

$$y = \gamma_0 + X^s \gamma + \varepsilon \tag{3.4}$$

where X^s is an $n \times 6$ matrix whose columns X_i^s are standardized HPM observations, $x_0, x_1, ..., x_i$ respectively. γ is a column vector of size 6 and its entries are coefficients $\gamma_0, \gamma_1, ..., \gamma_i$.

Our goal is to transform the model explanatory variables, X^s, into a new set of uncorrelated variables, which are the principal components of the correlation matrix $X^{s'} X^s$. We then remove the components that explain the least amount of variance.

Coefficients of Principal Components					
1	2	3	4	5	6
0.17	0.68	0.00	0.10	0.02	-0.71
-0.57	0.18	-0.02	-0.25	-0.77	-0.02
-0.55	0.08	0.01	0.81	0.16	0.06
0.14	0.47	-0.72	0.02	0.00	0.48
-0.56	0.18	-0.02	-0.52	0.62	-0.02
0.12	0.50	0.69	-0.01	0.00	0.51
% of Variance Explained					
47.61	33.44	15.77	2.75	0.40	0.04

Table 3.4: Principal components. Table shows the coefficients of principal components and the amount of variance explained. Principal components sorted in decreasing order, wrt. variance explained.

By doing so, we can decrease the effect of multicollinearity, and therefore narrow the confidence interval in our coefficient estimates.

As $X^{s\prime}X^s$ is a square matrix, one can decompose it into its eigenvectors V_i and eigenvalues λ_i such that $(X^{s\prime}X^s - \lambda_i I)V_i = 0$, where I is the identity matrix. The eigenvectors are orthonormal, that is $V_iV_j = 0$ for $i \neq j$. Let eigenvector matrix be $V = [V_1, V_2, ...V_6]$. Again, as eigenvectors are orthonormal, VV' gives the identity matrix I. Thus, one can rewrite Equation 3.4 as:

$$y = \gamma_0 + X^sVV'\gamma + \varepsilon$$

One can substitute X^sV with Z and $V'\gamma$ with θ:

$$y = \gamma_0 + Z\theta + \varepsilon \tag{3.5}$$

The normalized eigenvectors V_i of squared matrix $X^{s\prime}X^s$ give the coefficients of principal components. In other words, the principal components matrix Z is equal to the product of parameter observations X^s with eigenvector matrix V. After computing the principal components of the data, one can remove the multicollinearity by discarding the components that account only a fraction of the total variability. The penalty however is a decrease in model accuracy in exchange of an increase in confidence level of model parameter estimations. The eigenvalues show this variation. Let V_i is an eigenvector and λ_i is the corresponding eigenvector. The ratio $\lambda_i / \sum_1^6 \lambda_k$, gives the amount of variation that the corresponding principal component explains. Figure 3.4 shows the principal components of the data and the amount of variance that they explain (the largest first). As the figure suggests, the first three principal components account for more than 95% of model variability. Hence, we retain the first three components and discard the rest.

Next, we do regression using the principal components Z and compute the coefficients of principal components, that is the column vector θ. The estimated regression coefficients are free from the correlation (and thus, are more stable). Using θ, we compute the coefficients of standardized HPM counters, γ:

$$\gamma = V\theta$$

However, the γ gives the coefficients of the standardized HPM counters. Finally, as described in [52], We transform the coefficients back to the original HPM variables.

Computation Energy Consumption Models			
Description	MPCA Coefficients	MMISS Coefficients	MCPI coefficients
Constant	6.6	1.90	1.90
CPI	2.8	5.74	5.74
Inst. Miss	2814.5	180.0	−−
Inst. Not Dlvrd	−18.6	−−	−−
Data Stalls	3.05	−−	−−
Inst. TLB Miss	83039.4	−−	−−
Data TLB Miss	1055.8	−−	−−
R^2	0.99	0.99	0.99
Average Error	12.9%	5.81%	5.88%

Table 3.5: Coefficient and fit statistics for improved models.

We do this using:

$$\alpha_i = \frac{\gamma_i}{s_i}, 1 \leq i \leq 6$$

$$\alpha_0 = \gamma_0 - \sum_{i=1}^{6} \frac{\gamma_i \bar{x}_i}{s_i}$$

We call this model *MPCA*.

We develop MMISS and MCPI using the same method that we applied previously in Section 3.3. Table 3.5 summarizes the new models and their coefficients. As expected, the removal of three principal components reduces the accuracy of *MPCA* model, and consequently its error rate increases to 12.9%. However, this model enables us to extract information about the impact of each component in the model. The MPCA coefficients provide significant insight into memory and CPU energy consumption. We find that, (1) an instruction miss is approximately 1000 times more expensive than a clock cycle, (2) a data stall cycle (a clock cycle where pipeline stalls and waits for data) is slightly

more expensive than an instruction execution cycle (probably due to memory access activity), and (3) an instruction TLB miss is 75 times more expensive than a data TLB miss. However, the most interesting coefficient is instructions not delivered, which has a negative coefficient. A negative coefficient here does not mean a flaw in the methodology, but it indicates that the CPU (and memory) energy consumption drops below average level, which is the sum of constant factor, CPU clock counter and other events that the power model includes. This indicates that the CPU applies several techniques such as clock gating to reduce its energy consumption once pipeline is stalled.

The MCPI and MMISS model perform similarly and both are more accurate than MPCA. However, the t-statistic for the IMISS event is only 0.49 which indicates that this parameter is not significant. We therefore, remove MMISS from our model set.

The results show that CPI is a significant metric in the estimation of energy consumption. This is similar to the findings of prior study [7, 12]. Section 3.5 evaluates these models using a reference benchmark suite.

3.4 Modeling Communication Energy Consumption

We next introduce our model for wireless interface cost. This model is independent from the computation model to enable portability, i.e., we can swap the model for others for comparison or to improve accuracy. We combine the models via arithmetic

addition of the two estimates. Modeling the network interface is more challenging than modeling the processing unit because the network interface is significantly impacted by external effects such as RF interference, network congestion, asymmetric links due to badly calibrated hardware, etc. We do not consider these conditions since our goal is to explore the challenges of combining software counters with hardware counters.

As we did for the computation model, we employ a wide range of empirical observations from benchmarks to develop our communication model. The wireless network includes a set of 6 hosts, including PDAs and laptop computers. The network load varies from idle to a few megabits/second and is susceptible to interference from two separate wireless networks. Wireless speed rate is fixed at 11Mbits/sec.

The communication model is a linear parametric function like the computation model, and has four parameters: transmit bytes (TXB), receive bytes (TXB), transmit packets (TXP), and receive packets (RXP). The model uses these parameters as follows:

$$E_n(Joules) = TXB\beta_1 + RXB\beta_2 + TXP\beta_3 + RXP\beta_4 + K$$

The communication training benchmark suite considers three different scenarios: (i) upload heavy communications (ii) download heavy communications and (iii) almost symmetrical, mesh type communications. For the first two scenarios, we use the scp benchmark. To collect behavior from the symmetric communications, we use the netpipe benchmark to generate network load. Typically, netpipe transfers are ping-pong

Communication Energy Consumption Model			
Coef.	Description	Bytes+Packets	Bytes
TXB	TX bytes	2.40×10^{-6}	6.29×10^{-6}
RXB	RX bytes	-4.78×10^{-7}	-1.69×10^{-6}
TXP	TX packets	-2.90×10^{-3}	
RXP	RX packets	5.50×10^{-3}	
K	Constant	2.37×10^{-2}	2.00×10^{-1}
R^2		0.972	0.796
Average		26.9%	208%
95th Percentile		59.6%	470%
Wireless Idle Power		562 ± 146 mWatts	

Table 3.6: Communication energy model. The energy consumption of wireless card as a function of transferred bytes and packets.

like, it transfers one packet to a server and receives another packet before continuing. This forces the network to transmit every single packet, without opportunity to stream multiple small packets together. Netpipe also exposes idiosyncrasies that result from the internal hardware buffer, by re-evaluating each packet size using a constant perturbation factor. There are four transfer size categories: (1) small: < 100 bytes; (2) medium: 100 to 1000 bytes; (3) large: 1000 to 4000 bytes; and (4) very large: 4000 bytes to 200KB. We repeat each transfer 100 times for the first three categories and 10 times for the last category.

We consider two different models, one that considers both bytes and packets transfered and one that only considers bytes transferred. We refer to the former as (Bytes+Packets) and the latter as (Bytes). We present the LSQ coefficients for both models as well as the fit statistics for the training data set for the selected intervals in Table 3.6.

Both models exhibit much higher error rates than those from the computation model. The error is due to the difficulty of capturing external effects. However, these results are for energy consumption of the wireless card only. The evaluation section discusses benchmarks that perform both computation and communication. The error for the latter will impact overall estimation depending on the amount of communication performed by the application.

Interestingly, these results show that it is very important to consider both packet count and bytes transfered to produce an accurate model. By extending the byte model to include the packet counts, we improve the error rate of the model by almost an order of magnitude. The low accuracy of byte model reflects the non-linearity between transfer sizes and packet sizes. Small packets have a disproportionately large overhead due to protocol headers.

3.5 Validation

Here, we evaluate the efficacy of the proposed techniques. For comparison purposes, we use the closest published energy consumption model, that was described in [12]. We refer to this model as MCPUMEM.

MCPUMEM was developed for the same Intel XScale CPU and memory configuration and uses similar HPM-based techniques. One primary difference is that

MCPUMEM is limited to modeling and measuring memory and CPU energy consumption; it does not consider the full system. In contrary, our measurements and model include all I/O devices and other system components that exist on the embedded device. Thus, some divergence in the results is normal. However, the comparison of the two models are extremely useful in understanding and visualizing modeling challenges.

To correct and better understand the behavior of MCPUMEM, we (1) add the idle power consumption of wireless card to MCPUMEM (a constant factor), and (2) analyze MCPUMEM's mean error and its deviation. Deviation indicates how much the mean prediction error varies from one benchmark to another. As mean error can be affected by the idle energy consumption of hardware components, the deviation of the estimations better describes the quality of model. The following subsections discuss the models in detail.

3.5.1 Computation Model

Figure 3.1 shows the error rate for the reference benchmark set for proposed models. These models include MLARGE – the original model without the removal of multicollinearity; MPCA – the original model that uses principle component analysis for multicollinearity removal; MCPI – the original model with HPM metrics CPI; MCPUMEM – the CPU-only HPM-based model. MCPI and MLARGE perform similarly with an average error of 7%. The error is slightly lower for the MediaBench

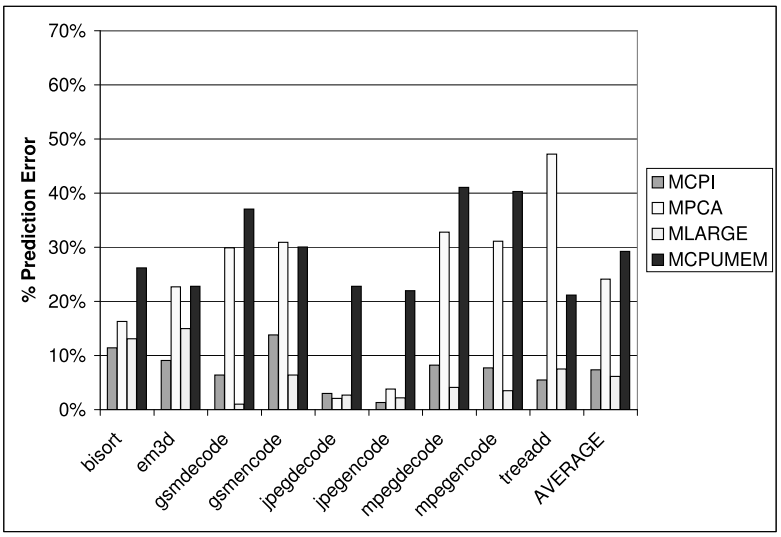


Figure 3.1: Error rate for the computation model.

benchmarks than for the Java benchmarks, this is because the behavior of the Java benchmarks is much more variable due to benchmark activity, garbage collection, class loading, interpretation, and other factors.

MPCA produces an average error of 22% making it unsuitable for energy estimation of computation-bound tasks. As explained previously, removing some principal components can reduce model accuracy while improving the parameter estimates. On the other hand, MPCA error rate is particularly low for two benchmarks, jpegencode and jpegdecode. These two benchmarks are less data intensive and more processor bound than the other benchmarks. Collected HPM data indicates that jpeg executes one instruction per data stall. The MPCA is particularly successful in modeling processor than the memory –the lack of a memory access counter reduces its memory model-

ing accuracy. However, since we remove some principal components from the model to improve model coefficient estimations, MPCA looses significant information about memory access cost.

MCPUMEM produces an average error rate of 30%. Moreover, its error rates vary from 20% to 40% across benchmarks, with a standard deviation that is twice that of the other models. The reason for this is that MCPUMEM is designed to estimate the power consumption of the CPU alone and not the full system.

A significant result is the success of MCPI model. The MCPI has an error rate close to the MLARGE model, using a single model input -the clock cycles per instruction. On XScale, one can gather this information accurately by reading the two hardware performance monitors; the clock cycle counter and the instruction counter. This can lead to a run-time power model since an ordinary XScale is already capable of collecting this information using its two HPM counters.

3.5.2 Communication Model

We next integrate the computation and communication model and evaluate the accuracy of the combined model. We estimate the energy consumption using $E_t = E_l + E_n$ where E_t is the total energy consumption, E_l is the computation model output and E_n is the network model output. We use the names of the computation models that was discussed in the prior section to identify the integrated models in this section.

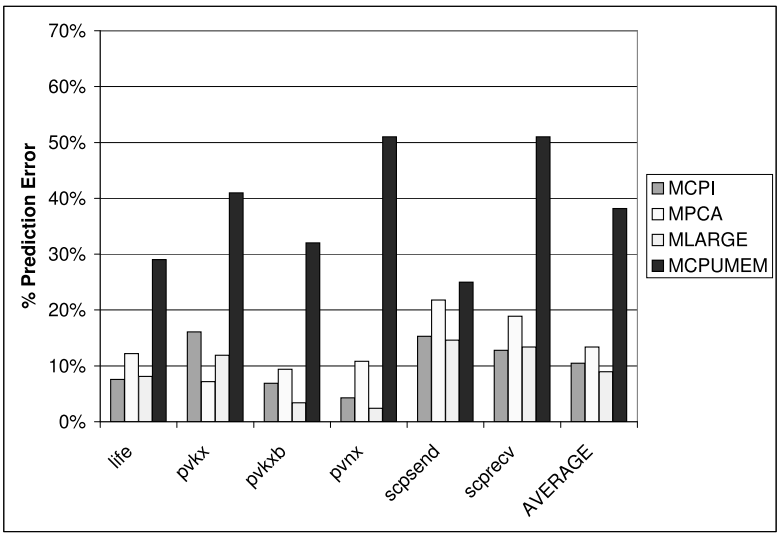


Figure 3.2: Error rate for the communication model.

Figure 3.2 shows the percent error for all the models. MCPI and MLARGE perform similarly with an average error rate of 11%. The error rate is 38% on average for MCPUMEM.

Interestingly however, in communication dataset the MPCA is as accurate as the other models, with an average error of 13%. The prediction error of MPCA is below 25% for all programs. MPCA is significantly different than all other models. As Table 3.5 shows, MPCA gives much higher weight to instruction pipeline events (TLB miss, instruction not delivered, etc) than the other models. Due to frequent I/O between wireless card and CPU, such events play a significant role in describing program energy consumption. Hence, MPCA is important as it addresses the deficiencies in other models.

3.6 Why Linear Regression?

In this chapter, we limit ourselves to a linear model to explain the relation between power consumption and hardware (and software) events. In other words, given the events x_i and energy consumption E

$$E = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_k x_k \tag{3.6}$$

we develop our models based on the assumption that the parameters β_i are linear. Here, we do not consider non-linear models. Our reasoning is two-fold. First of all, a large body of extant work has already demonstrated highly accurate linear models for various hardware components including memory [39], CPU [7, 38, 78, 39, 35, 12], and I/O [46]. Second, as we show in Section 3.3, the high R^2 of our model is a strong indication that our linearity assumption is true.

An alternative way of estimating linear model parameters from observed data is maximum likelihood [9]. The goal of maximum likelihood is to find the parameters that make the observed data most likely. This method assumes a fixed observation vector x, and defines a function $L(\theta|x)$ which shows the likelihood of parameters θ given the observations x. A maximum likelihood estimator for $\theta(x)$ is a value that maximizes the value of $L(\theta|x)$ for the given observations x.

To explain this better, imagine a coin toss experiment using a biased coin. Here, let θ is the probability of having heads as experiment outcome. If we observe 60 heads

and 40 tails in 100 experiments, we can find the likelihood of having model parameter $\theta = 0.5$ (a fair coin) using binomial distribution:

$$L(\theta = 0.5|x) = \frac{100!}{60!\,40!}\,0.5^{60}\,0.5^{40} = 0.01$$

Even though maximum likelihood is known to perform well when the sample size is large, its estimators are biased when sample size is small. Furthermore, maximum likelihood is rather complicated when multiple parameters are involved in the model and an efficient run-time implementation of it is not obvious to us (which is our major goal as we discuss in next section), given the constraints of our target platform. For this reason, we do not evaluate this approach further in this dissertation, and leave it as a future direction.

3.7 Related Work

This chapter discusses full system energy consumption modeling for low power devices. It models the Stargate sensor network gateway as a case study. The models it proposes use hardware and operating system monitors to estimate the power consumption of a sensor network gateway. The work most related to this is on HPM-based models for CPU and memory energy estimation [7, 38, 78, 39, 35, 12]. In recent work, Bircher et al. [7] presents a power model for the Pentium-IV class of processors. They develop their power model using least squares regression (LSQ) [23] and show that two

hardware counters are enough to model energy consumption on their target architecture. However, they do not consider memory, and memory bus in their study.

In embedded systems, the most similar study to our approach is by Contreras et al. [12]. In this work, the authors use LSQ to develop a power model for an Intel XScale processor attached to a development board. Using this model, the authors are able estimate CPU energy consumption with a 4% error rate. However, their efforts to construct a memory power model did not perform as well due to the lack of hardware counters in the CPU that count memory events. Here, we compare this HPM model to estimate full system energy consumption by employing software counters. This model considers a larger set of components including memory and memory bus and demonstrates a lower error rate.

3.8 Summary

This chapter presents a system for estimating full-system power consumption of Crossbow Stargate sensor network device. It couples statistical techniques that employ empirical data from hardware and software performance monitors to model the computation and communication of executing tasks. The results indicate that metrics like instruction execution rate, memory access rate and data transfer rate are quite effective in predicting energy consumption of the full system. It finds that the model that

predicts energy consumption does not necessarily have to be large for better precision. Furthermore it demonstrates that larger models are more likely to suffer from higher error rates due to multicollinearity -the linear dependencies between model parameters. Multicollinearity reduces coefficient estimation stability and prevents extracting useful information about the contribution of the model components. Finally this chapter shows that principle component analysis can reduce multicollinearity in the model data at a cost in accuracy.

Chapter 4

Predicting Energy Consumption at Run-Time

The previous chapter claimed that a run-time power measurement system is necessary for new power optimizations, and it explored the challenges of modeling energy consumption using hardware performance monitors and linear regression. This chapter proposes a run-time power estimation mechanism for low-power, resource-restricted embedded computers. Section 4.1 describes extant power measurement methods and discusses their advantages and disadvantages. Section 4.2 describes the high-level ideas behind the proposed power profiling system, Section 4.3 describes evaluation methodology, and Section 4.4 validates it on a popular embedded platform. Section 4.5 discusses why we choose our model, and compares it to alternative approaches. Section 4.6 presents related work. Finally, Section 4.7 gives a summary and concludes.

4.1 Extant OS Support For Measuring Energy Use

Current operating systems provide little support for run-time energy profiling. Usually, a simple operating system interface provides access to a device that controls the battery charge/recharge cycles. This device has a voltage sensor for continuously monitoring charge level to prevent any over-charge. By comparing the output of this sensor to a set of voltage measurements collected at known charge levels, operating system can roughly estimate the current battery charge and the rate of energy use. However, this information is too coarse and imprecise for almost any kind of energy profiling.

A recent, second-generation battery monitoring unit [13] (aka BMU) enables much higher fidelity. The battery monitoring unit continuously monitors the voltage drop across a high-precision current sense resistor to compute the current flow. The voltage drop, when divided by the value of the sense resistor, gives the instantaneous current flowing into and out of the battery. The battery monitor interpolates these measurements over time to compute the net charge that is left in the battery. An internal accumulator holds the result of this computation. By reading the accumulator state before and after the execution of a software task, a user application can compute the rate of energy use.

Even given these significant improvements in energy measurement technology, they still only enable coarse-grained and inaccurate readings. The latency of the system

makes measurement possible at only large intervals (10s of milliseconds), precluding our ability to attribute energy consumption to only behaviors and tasks with long duration (on the order of seconds) [13]. Moreover, the low resolution of the A/D converter and the limited width of the internal registers restrict sample precision. In addition, the relatively long, slow, serial path between the application, through, the operating system, to the battery pack, prevents us from extracting battery levels in real time (e.g. the instant a task completes) and thus, increases the granularity and decreases the accuracy of the energy measurements that we make.

In order to better understand the capability of these devices, we analyze one such advanced battery monitoring unit. Using [13], we can compute the energy consumption from time t_1 to time t_2 by reading the accumulator register and battery voltage. More specifically, let (v_1, ac_1) and (v_2, ac_2) be voltage and **accumulated current** readings at time t_1 and t_2. The energy consumption at [t_1, t_2] is:

$$E = (v_1 + v_2)/2 \times (ac_1 - ac_2) \times 3600 \text{ sec/hours}$$

This equation does not include time since ac is the accumulated current and not the average current. The multiplier, 3600 sec/hours, converts the result to microjoules.

In [13], each current reading may be upto 0.25 milliAmpere-hours (mAh) different than the actual charge in battery. 0.25 mAh is equal to the aggregated electrical charge when a current of 0.25 milli-amperes passes through battery terminals for a period of one hour. In terms of energy, this is equal to 3.24 joules (i.e. $3600 \times 0.25 \times 3.6$) at the full

battery charge level (3.6 Volts). This is a very small error for battery lifetime estimation but it is rather high for software profiling. Indeed, a fully loaded iPAQ uses less than half of this energy in a second. Thus, only the tasks that are extremely long (i.e. tens of seconds) can be power-profiled using a battery monitoring unit. However, even then, the lengthy communication latency does not allow collecting this information accurately.

As a workaround, extant systems use execution time and CPU cycles to estimate energy cost. These metrics can be measured quickly and precisely using operating system clock. However, execution time and CPU cycles are hardly correlated to power consumption. For example, on a Pentium-IV processor, the power consumption varies from 30 Watts to 90 Watts depending on the type of instruction [35]. Energy consumption also changes depending on the processor voltage, power/performance settings of other components, etc. Thus energy should be measured directly, whenever possible.

4.2 Proposed Run-time Energy Prediction Mechanism

Figure 4.1 reveals the high-level overview of proposed power profiling mechanism. The proposed mechanism contains a model that maps hardware and software counter values into power values. The model is adaptive; it can continuously improve itself by monitoring its error rate. Although a static method is more desirable from a computational point of view, it cannot adapt to the dynamics of the run-time power behavior.

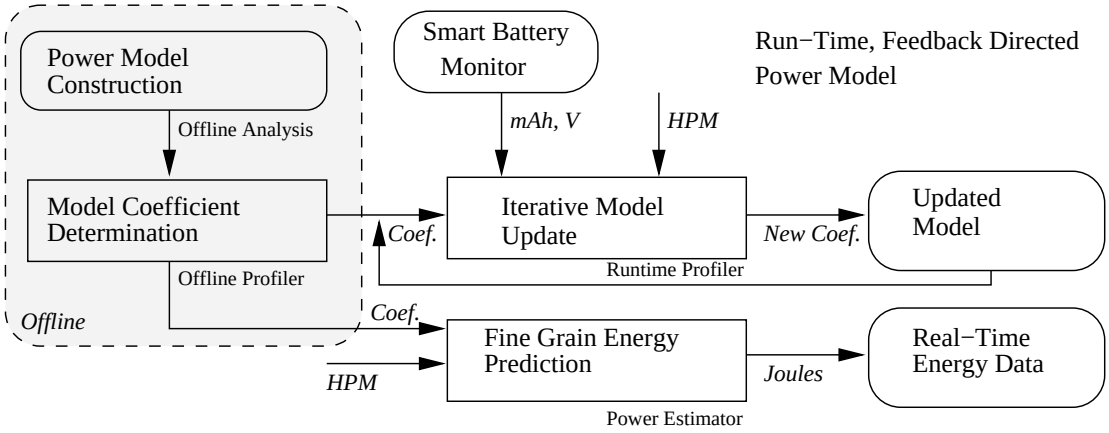


Figure 4.1: Block diagram of proposed run-time power prediction system

A static model may produce incorrect results if there is a large change in workload characteristics, or if hardware power/performance settings are altered. To attack this challenge, we propose a progressive model; one that can monitor its prediction error using feedback from battery monitor and can adapt over time.

The heart of our system is the run-time profiler. The profiler polls the battery monitoring unit that we described in Section 4.1 and accumulates values from software and hardware counters periodically. After each period, the profiler updates the model parameter coefficients iteratively to improve its prediction accuracy and to adapt to potential changes in power behavior. We call each such period a *model update period*.

Due to its stability, robustness, adaptivity and modest computational demand, we use *recursive least squares linear regression with exponential decay* [85] (i.e. *RLS-ED*) to update model coefficients iteratively. The RLS-ED is a recursive implementation of

the well-known least squares linear regression. Using a decay factor, it exponentially reduces the weight of the oldest measurements. With the measurements u_k at time k, and the decay factor γ, RLS-ED weights the measurements using:

$$u_k + \gamma u_(k-1) + \gamma^2 u_(k-2) + ... + \gamma^k u_0$$

The γ adjusts the adaptiveness of the algorithm ($\gamma \leq 1.0$). A smaller γ means the model is more responsive to changes in the input data but less resilient to noise. In Section 4.4.1, we discuss the effect of γ on regression accuracy and stability.

Internally, RLS-ED maintains a matrix of size $n \times n$ (n is equal to the number of model parameters) to retain the state information between each iteration. Each RLS-ED iteration involves eight matrix multiplications, each of which requires approximately one hundred floating point operations when $n = 2$. Even though this may not be a significant cost on high-end machines, these floating point operations can consume significant resources on many resource-constrained platforms. To reduce this cost, we explore policies to reduce RLS-ED iteration frequency, in the next section. In addition, since the asymptotic complexity of the algorithm is O(n^3), the model must be as small as possible to keep n small and the computational cost of the algorithm low. We discuss RLS-ED execution period (i.e. model update period) and execution cost in next section.

The *power estimator* uses hardware and software counters and model coefficients to estimate corresponding power consumption. To estimate the energy consumption of a task (i.e. here, we use intervals of 10 million instructions instead of software tasks), a

software component has to collect the values of these counters immediately before and after the execution of the task. This can easily be done by using and extending portable middleware systems like PAPI [58]. Since the LSQ models that we use have only a few inputs, each estimation in our model requires fewer than 10 floating point operations.

The *offline profiler* is the only optional component of our system. As the RLS-ED algorithm is recursive, it requires an initial state to start its iterations. Without the existence of an offline profiler, the error rates can be high until the algorithm reaches a stable state. While the offline profiler has the potential to improve model accuracy considerably, the extra effort associated with profiling makes it undesirable. The forthcoming section discusses the extra accuracy such a profiler can provide.

4.3 Evaluation Methodology

Our approach to model evaluation is empirical. We collect real power measurements using the setup and benchmarks described in previous chapter (i.e. Section 3.2), inject error to them as would be expected in real battery monitor measurements, and evaluate the models on these data by varying their parameters. We then compare prediction results to real measurements. This section discusses the power models. The next one describes the evaluation in depth.

	Computation	Communication
Compact	Cpu Cycles Data Stalls	Cpu Cycles Tx Bytes Rx Bytes
Complex	Cpu Cycles Inst. Miss Inst. NDlvr Data Stalls Inst TLB Miss Data TLB Miss	Cpu Cycles Tx Bytes Rx Bytes Tx Packets Rx Packets

Table 4.1: Input variables in derived power models

Table 4.1 shows the two models that we derive. The first one, to which we refer as complex model, has a computation (E_c) and communication (E_n) subcomponent. The components are defined as:

$$E_c(Joules) = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_6 x_6$$
$$E_n(Joules) = \alpha_1 x_1 + B_{tx}\beta_1 + B_{rx}\beta_2 + P_{tx}\beta_3 + P_{rx}\beta_4 + K$$

where x_i's are *core clock cycles, instruction cache misses, instructions not delivered, data stalls, instruction TLB misses,* and *data TLB misses,* respectively. The α's are computed as described in Section 3.3. In complex communication model, (B_{tx}), and (B_{rx}), specify the transmitted and received bytes as we defined earlier. P_{tx} and P_{rx} are the transmitted and received packet counts, respectively.

The computation model is same as the large model in previous chapter. We choose this since it has the lowest error across all benchmarks. The communication model is

different, however. Unlike the previous static model, the recursive, RLS-ED model is very sensitive to any increase in parameter size due to O(n^3) computational cost. Thus to limit parameter size, we define the communication model as a combination of MCPI (which was shown to be the second most effective model) and packet/byte counters.

The second model, to which we refer as compact model, also consists of computation and communication components. The computation model estimates the energy consumption of tasks that execute without any communication or any significant access to persistent storage. This model includes three parameters:

$$E_c(Joules) = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 \tag{4.1}$$

where x_i's are core clock cycles and data stalls, respectively. The communication model uses two *software performance counters*: transmit bytes (B_{tx}), receive bytes (B_{rx}) and one hardware counter, core clock cycles. Again, we do not include other HPM events to reduce the cost of RLS-ED iterations. The compact communication model is:

$$E_n(Joules) = \alpha_1 x_1 + \beta_1 B_{tx} + \beta_2 B_{rx} + K \tag{4.2}$$

Here, α_1 is the weight of core clock cycles and β's are weights of transmit and receive bytes. At present, the models do not incorporate low level card state (idle, etc.) information, as we favor a simpler model at the expense of a potentially higher error rate.

Computation Benchmark Set		Communication Benchmark Set	
Application	Time (s)	Application	Time (s)
gsmdecode	1.0	game of life (MPI)	9.02
gsmencode	1.1	pvkxb (MPI)	35.2
jpegdecode	5.4	pvnx (MPI)	37.78
jpegencode	17.1	pvkx (MPI)	69.36
mpegdecode	72.9		
mpegencode	91.7		
em3d (Java)	12.1		
bisort (Java)	20.4		
treeadd (Java)	3.8		

Table 4.2: Prediction benchmarks. The set to the left are to model/evaluate computation; the rest are for communication.

We validate our model using the benchmarks described in Section 3.2. We only use the benchmarks in the reference set; Table 4.2 reminds these for convenience.

4.4 Results

This section evaluates the models that were presented. It explores the performance of models in terms of accuracy, under various factors, such as update rate, mesaurement error and agility (RLS-ED decay factor). In addition, it discusses the execution cost of both run-time computation, and battery monitor access.

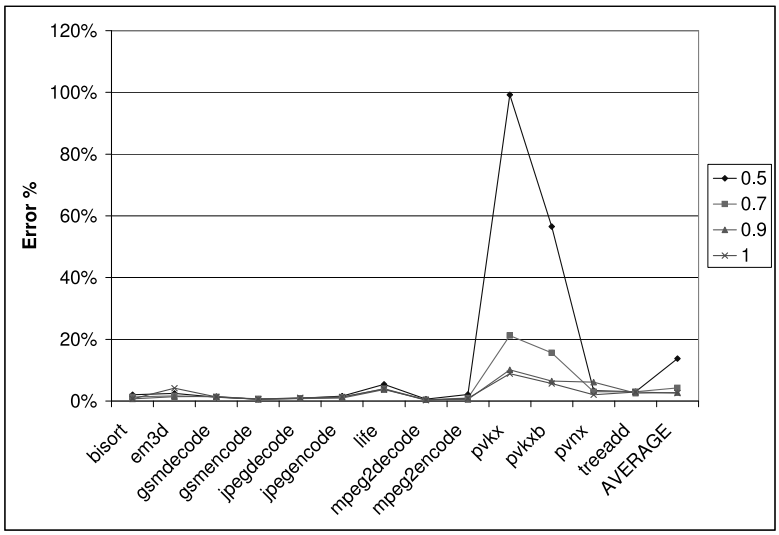


Figure 4.2: Decay factor vs. accuracy. A lower γ gives exponentially greater weight to most recent data. However, this makes the algorithm more vulnerable to noise. The results show that RLS-ED algorithm works best when γ is between 0.9 and 1.0.

4.4.1 Decay Factor vs. Accuracy

We first explore the relationship between decay factor, γ, and accuracy. In this experiment, we do not consider execution overhead, hence, we update model coefficients after each interval. We initialize the coefficients differently for computation and communication models. For the former, we use the values described in previous chapter (i.e. Chapter 3). For the latter, we monitor a secure file transfer (i.e. scp) of 17MB file across the wireless network multiple times and run the offline profiler to generate the initial values.

Figure 4.2 shows evaluation results for $\gamma = 0.5, 0.7, 0.9$ and 1. When the decay factor is 1, the algorithm becomes ordinary recursive least squares regression and does

not decay any of the previous values. Conversely, when the decay is 0.5, the algorithm effectively remembers only the most recent four measurements. For each benchmark, we show the average estimation error, which we compute using $|measured - estimated|/measured \times 100$. The error rates vary from 0.5% to 10% in general, giving an average error of 2.6% for $\gamma \geq 0.9$, and increasing up to 13.7% when $\gamma = 0.5$.

As expected, the error rate is higher for the communication benchmarks. We have encountered a few atypical cases and in one of the benchmarks the error rate was equal to almost 100%, which means the predictions were off by a margin equal to the real value. These atypical cases were specific to low decay factors and to the pvkx benchmark. Pvkx is an MPI program that includes a lot of short communication and computation phases. These phases generate sudden, transient changes in program behavior. When the decay factor is very low, RLS-ED remembers a very short history and reacts much faster than necessary, generating erroneous estimations. In other cases, the error rates are much lower. Overall, moderate decay provides the best result.

In Figure 4.3, we compare the adaptive model, $\gamma = 0.9$, to the case when the model is completely static. The dynamic model provides much lower estimation errors in general (2.6% to 5.6%). The only benchmark for which the dynamic model generates higher error is pvkxb (6.5% vs. 4.7%). Pvkxb is similar to the pvkx benchmark, however, it is much shorter. As a result, pvkxb offers very few adjustment opportunities to RLS-ED algorithm.

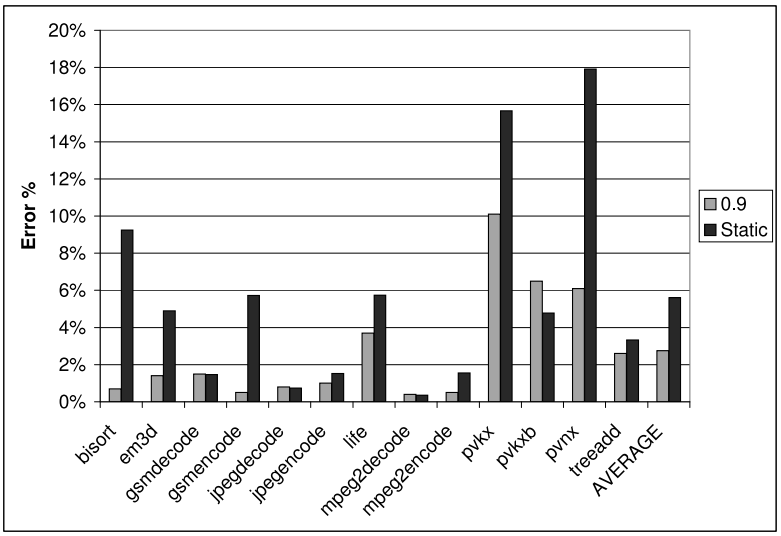


Figure 4.3: Static vs. adaptive models. Figure compares the error rate of the adaptive model with a static one. The adaptive model generates better results in almost every benchmark.

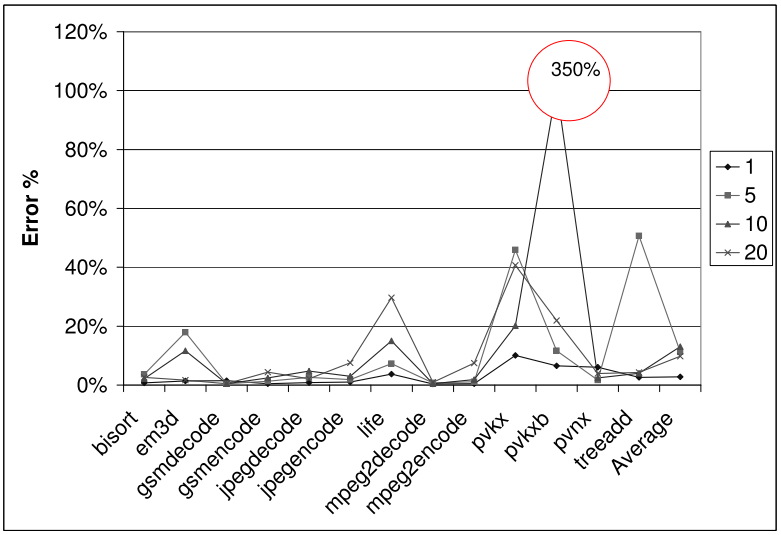


Figure 4.4: RLS-ED update frequency vs. accuracy. Figure compares the error rate of 4 update policies. 1 means that the system updates the model after every interval, 5 means that the system accumulates the monitored statistics for 5 intervals before updating, etc.

4.4.2 Update Period

Figure 4.4 shows the error rate when RLS-ED algorithm runs *infrequently* to reduce its computation overhead. The update policy combines ρ consecutive intervals into a single super-interval and updates the model once for each. However, we estimate energy consumption after every interval as we did in our prior experiment. We experiment with $\rho = 1, 5, 10$ and 20 and use a decay factor of 0.9.

As expected, the average error rate across all benchmarks increases as we increase the value of ρ. However, the increase is *not linear* and produces surprising results, especially for the network applications. For pvkx and treeadd, the error rate exceeds 50% when $r = 5$ at a single point. For pvkxb, which was our most challenging benchmark in many cases, the error rate tops 350% when $\rho = 10$ and decreases when we increase the r to 20.

4.4.3 Benefits From Offline Profiling

We next investigate the efficacy of using the offline profiler to reduce model error rate during the initial warm-up period of the RLS-ED algorithm. Figure 4.5 shows the results across benchmarks for $\gamma = 0.9$. The light colored bars show the error rate when we use an offline profiler, the dark colored bars show the error rate when we do not. For the offline profiler, we determine the coefficients as we outline in Section 4.4.1. In the absence of the offline profiler, we initialize all the coefficients to 0.

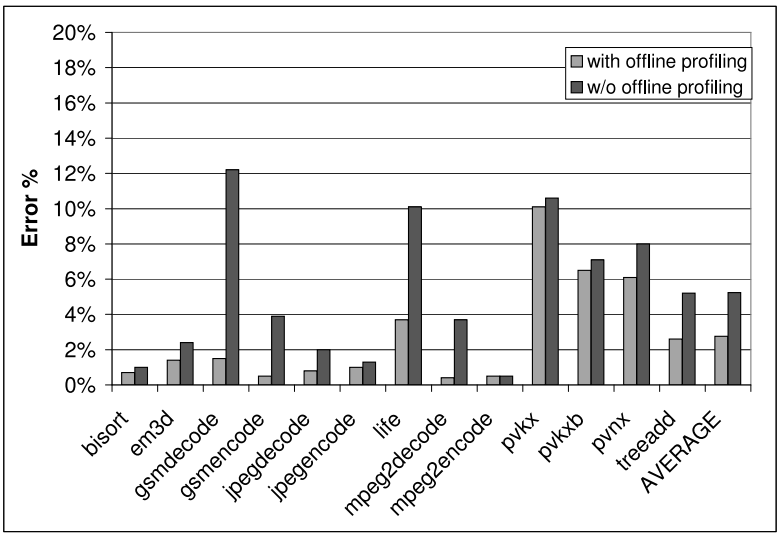


Figure 4.5: Benefit from an offline profiler. The offline profiler reduces error rate 2.5% in average.

The offline profiler reduces the error rates for all cases. The benefits are more clear for the shorter benchmarks such as gsmdecode, gsmencode, and life. The offline profiler only marginally effects the network benchmarks such as pvkx, pvnx and pvkxb. In contrast to the life application, these three benchmarks tend to transfer smaller amounts of data between their computation period, and are more susceptible to variations in network latency. Overall, profiling reduces error rate from 5.2% to 2.7%.

4.4.4 Battery Monitor Error Rate vs. Accuracy

A novel feature of our proposed model is the use of the battery monitor as feedback to adjust the model coefficients at runtime. The internal BMU, however, is imprecise, and introduces a much higher error rate than that of our external, high-precision, equip-

ment. In our target platform, the energy readings are within 4.88mVolts and 0.25mAh (milliampere-Hours) of real voltage and current flow of the battery pack. We assume that the battery voltage stays stable between two readings because of the short period, hence we only consider the current flow measurement errors.

Two other factors, although not directly related to the accuracy of BMU readings, significantly influence our design. The first factor is related to the BMU access overhead. The BMU and CPU is connected through a serial, *one-wire* link and frequent accesses incur an overhead. The second factor is the computational cost of the RLS-ED algorithm. Therefore, we combine ρ *consecutive intervals into a single super-interval* and update the model once for each. In this section, we evaluate several factors of ρ and algorithm accuracy.

The BMU datasheet [13] does not provide any details about measurement error distribution. In this case, we assume a uniform distribution such that the difference between real values and the observed values can be in the range $[-0.125, 0.125]$ mAh. To explore the effect of this error, we injected artificial error into the current flow measurements immediately prior to running the RLS-ED algorithm. We call this amount of error *1X precision*.

To capture future improvements in battery monitoring technology, We also investigate three other precision levels; *2X, 4X, and 8X*. The prefix before *X* is the ratio of reduction in error rate, for example, *2X* has an error range $[-0.062, 0.062]$ mAh. We

compare these results to a *precise* battery monitoring unit, which has a precision that is equal to that of our external measurement equipment (no artificial error).

4.4.5 Performance Of Complex Model

Tables 4.3, 4.4, and 4.5 show the results for three RLS-ED update periods, $\rho = 100, 200$, and 400, respectively. As the unit of ρ is instructions executed, the exact length of update period in wall clock time is somewhat arbitrary. However, in an Intel XScale CPU running at 400 MHz, the updates are separated by at least 10 seconds (much more in practice) when $\rho = 400$, and less for the other cases. In each table, we group the results by ρ and then divide each group into columns of precision. The X in the column header shows the precision level. ∞ means that the precision is equal to the external equipment. The tables show results for both the compact and the complex models.

We set the decay factor, γ, to 0.9. The offline profiler warms-up the coefficients by running a benchmark once, and then repeatedly runs the benchmark until we monitor at least 2000 intervals. By repeatedly executing them, we can monitor how the feedback and adaptiveness mechanism of the algorithm behaves even for the benchmarks that are shorter than one period.

As the results show, there is a large discrepancy between the *imprecise* (i.e. 1X to 8X) and the *precise* cases. When no measurement errors are present, the RLS-ED algo-

	1X	2X	4X	8X	∞
bisort	460.2	232.0	61.3	58.6	3.7
em3d	1004.4	498.7	184.7	126.2	5.4
gsmdecode	7.0	4.1	3.4	3.5	3.0
gsmencode	693.0	295.6	229.3	86.9	0.7
jpegdecode	79.9	107.8	31.6	9.9	1.4
jpegencode	45.8	19.3	14.0	6.1	1.2
life	149.9	72.3	40.7	19.9	4.3
mpeg2decode	18.2	14.2	5.7	2.4	0.4
mpeg2encode	30.8	21.1	11.9	5.0	3.7
pvkx	69.4	29.9	19.5	13.1	9.4
pvkxb	49.4	35.1	21.9	17.4	16.1
pvnx	24.0	29.3	13.2	6.9	5.8
treeadd	77.8	33.0	18.7	10.0	1.0

(a) Complex Model

	1X	2X	4X	8X	∞
bisort	5.9	2.8	2.3	2.4	1.8
em3d	165.8	130.1	64.8	114.6	2.7
gsmdecode	2.5	1.1	0.8	0.7	0.3
gsmencode	136.3	90.3	17.9	19.3	0.3
jpegdecode	87.4	23.6	23.0	9.9	1.0
jpegencode	32.5	10.4	7.4	7.1	4.1
life	145.3	50.2	19.6	23.3	4.7
mpeg2decode	10.7	2.8	1.9	1.5	0.3
mpeg2encode	16.9	10.7	6.1	5.0	2.0
pvkx	16.6	14.3	13.2	9.7	8.7
pvkxb	32.7	31.1	19.6	20.6	20.2
pvnx	13.1	7.2	3.6	2.5	1.1
treeadd	28.4	14.4	8.7	4.3	2.1

(b) Compact Model

Table 4.3: Comparison of model error rates, updating every $\rho = 100$ intervals.

	1X	2X	4X	8X	∞
bisort	1348.9	353.7	273.6	169.2	5.1
em3d	709.9	248.0	101.5	91.7	4.3
gsmdecode	7.0	7.1	6.3	6.0	5.9
gsmencode	728.6	315.3	183.5	91.3	1.0
jpegdecode	49.7	40.6	34.8	6.6	0.7
jpegencode	25.5	24.0	10.1	4.0	1.4
life	127.8	48.6	42.6	17.8	3.8
mpeg2decode	22.4	9.6	4.7	2.9	0.4
mpeg2encode	34.0	37.1	7.0	5.6	5.6
pvkx	66.7	19.6	14.7	12.5	8.2
pvkxb	33.9	28.2	7.9	8.1	6.8
pvnx	12.1	9.2	6.0	6.0	5.4
treeadd	91.1	42.6	15.8	12.1	1.4

(a) Complex Model

	1X	2X	4X	8X	∞
bisort	11.3	3.8	2.1	2.3	1.9
em3d	280.9	58.8	46.3	16.3	2.2
gsmdecode	2.9	1.4	0.9	0.7	0.5
gsmencode	298.4	118.6	49.1	20.5	0.4
jpegdecode	19.6	22.8	9.1	14.2	1.2
jpegencode	23.5	4.4	3.7	3.3	2.0
life	90.1	44.5	17.0	10.6	4.8
mpeg2decode	5.0	2.8	1.7	0.9	0.3
mpeg2encode	10.0	5.7	4.0	4.2	4.3
pvkx	40.1	12.2	14.1	9.8	7.2
pvkxb	51.4	22.0	10.2	7.8	6.6
pvnx	16.0	5.3	2.7	1.8	1.2
treeadd	45.6	11.5	7.7	3.3	2.1

(b) Compact Model

Table 4.4: Comparison of model error rates, updating every $\rho = 200$ intervals.

	1X	2X	4X	8X	∞
bisort	31.9	24.1	7.9	6.9	5.4
em3d	387.7	148.1	96.8	51.3	3.6
gsmdecode	12.6	12.2	11.8	11.7	11.6
gsmencode	1411.3	447.9	419.7	177.7	0.7
jpegdecode	45.8	13.6	21.2	5.8	0.7
jpegencode	24.7	10.9	5.1	3.9	1.6
life	106.0	16.6	8.1	15.1	3.7
mpeg2decode	3.1	0.8	0.7	0.6	0.4
mpeg2encode	26.8	17.4	6.2	5.1	4.0
pvkx	33.5	12.9	10.9	9.9	8.2
pvkxb	31.3	8.9	8.6	7.2	5.5
pvnx	6.7	6.4	5.7	5.8	6.0
treeadd	81.5	43.8	23.4	11.5	1.8

(a) Complex Model

	1X	2X	4X	8X	∞
bisort	7.8	4.6	5.6	3.3	2.9
em3d	91.9	79.2	8.5	7.8	2.5
gsmdecode	2.1	1.2	1.2	0.9	0.8
gsmencode	43.4	58.3	6.6	34.5	0.5
jpegdecode	18.7	47.6	25.6	12.0	0.9
jpegencode	20.2	5.7	4.8	6.0	2.9
life	88.7	7.2	29.2	9.3	4.4
mpeg2decode	1.9	1.0	0.5	0.4	0.3
mpeg2encode	11.8	8.4	6.0	5.6	5.4
pvkx	36.0	24.0	12.2	8.5	7.3
pvkxb	14.9	18.2	7.7	6.8	5.2
pvnx	3.7	5.7	2.5	1.5	1.3
treeadd	88.9	25.2	8.0	4.2	2.2

(b) Compact Model

Table 4.5: Comparison of model error rates, updating every $\rho = 400$ intervals.

rithm converges quickly, providing estimations that are within 10% of the real values. When measurement errors are present, the estimation error rates increase significantly. This increase is more apparent for some applications such as gsmencode and em3d. These applications have short, sudden changes in their energy consumption behavior. For instance, gsmencode is a very short benchmark with a very smooth execution pattern except the very first few intervals. During these intervals, the energy consumption increase sharply. When these spikes coincide with energy measurements, the RLS-ED detects an immediate increase in energy consumption and overestimates the model parameters.

The effect of ρ on the total system is less obvious, because a higher ρ imposes two different effects. First, as ρ increases, the relative magnitude of measurement errors asymptomatically decrease. This is a result of the constant error factor that the battery monitor imposes. For instance when precision is $1X$, the expected error rate is $(0.25)/E$. As a higher ρ means a larger observation period, E becomes larger and error rate becomes smaller. Second, a higher ρ means less frequent model updates, giving the model less chance to react when program behavior changes. Our results show that $\rho = 400$ is better, however, the best ρ varies from one benchmark to another.

4.4.6 Performance Of Compact Model

The tables above also show the same results for the compact model. The compact model error rate shows a significant improvement over the complex model especially when there are feedback errors. For example, when $\rho = 100$ and precision is $1X$ the average error rate is 208% for complex model and 53.3% for compact model. We find that the compact model error rate is less than 3% when there are no feedback errors.

The poor performance of complex model is a result of *multicollinearity*, which was described in Section 3.3.1. In the presence of linear dependence between the variables, the recursive estimates of the RLS-ED algorithm converges slowly and produces inaccurate parameter estimations [85]. The presence of errors in battery monitor readings and a large ρ further complicates the model and reduces the accuracy. The compact model, since it has fewer (and many fewer related) parameters, does not suffer from this phenomenon.

The results indicate that the accuracy of the algorithm is highly dependent on the feedback error rate. At the levels of precision available from BMUs in current devices (i.e. 1X) , feedback errors have a significant adverse effect on algorithm accuracy. However, once the precision levels improve to $8X$, or more, the increase in estimation accuracy improves the quality of energy estimates. When BMU precision level is 8X, the error rate drops to less than 8.0% for $\rho = 200$ and $\rho = 400$.

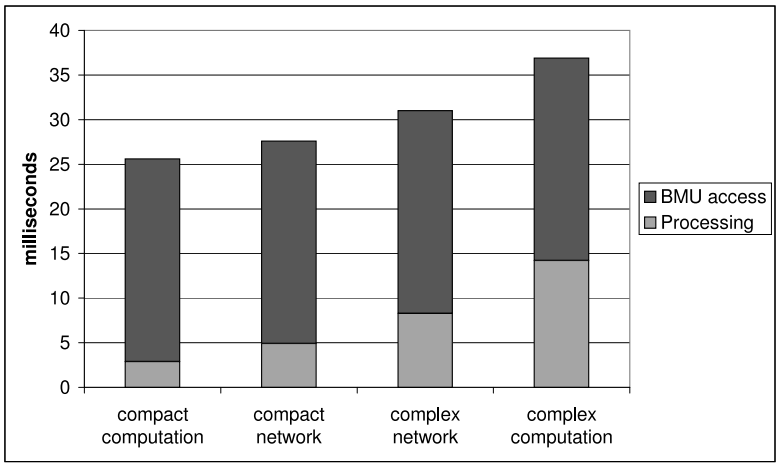


Figure 4.6: RLS-ED execution cost. The bars show the average CPU time used for each RLS-ED iteration.

We also find that the selection of ρ is an important factor in model accuracy. Some benchmarks, like em3d and gsmencode, are highly sensitive to the value of ρ. This is a result of short, sudden changes in program behavior (mostly during initialization) that coincide with the RLS-ED updates to the model. Even though we do not evaluate it in the scope of this paper, an application specific selection of ρ may provide better convergence in these cases.

4.4.7 Execution Cost

Figure 4.6 shows the cost of executing our model on our target platform. We implemented this model in C as a user-space application. The height of each bar shows the average execution time for a single iteration. The dark colored portion of the bars shows the BMU access time, including the cost of reading data from hardware to ker-

nel space and then transferring to user space. The BMU access time is in average 22.7 milliseconds, and same for all models. The light colored portion of the bars shows the RLS-ED execution time. For the compact computation model, we measured each iteration to consume 2.9 milliseconds of CPU time. The RLS-ED cost is proportional to the square of model parameter count, and increases up to 14.2 milliseconds for the complex computation model. However, these results show that the dominant cost is the battery monitor access time and not the RLS-ED computation.

4.5 Why RLS-ED?

To update linear model parameters at run-time, we choose recursive least squares algorithm because of its robustness, stability, adaptivity and modest computational demand. Our evaluation shows that this method continues to perform well even in the presence of model errors and parameter dependencies.

As Figure 4.1 shows, the recursive update forms a discrete feedback system which may also be seen as a low-pass filter [85]. When there is no exponential decay, each new measurement has gradually less effect on model update, as the model considers all the past measurements. The model converges fast when the model is stationary. In contrary, our system is not necessarily stationary. As the time passes, the energy consumption behavior of the system may change because of changes in workload characteristics

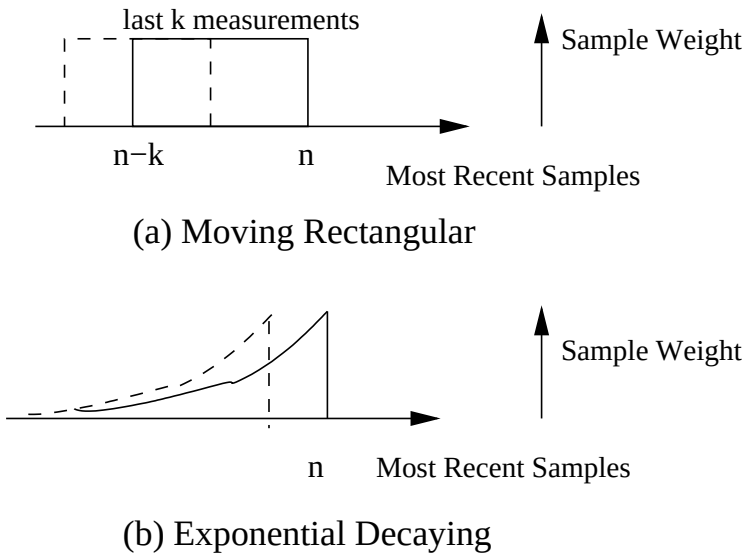


Figure 4.7: Recursive least squares memory shaping

(which we also want to capture). The recursive least squares algorithm provides two methods to handle time varying situations: a rectangular data weighting algorithm, and an exponential data weighting [23]. Figure 4.7 illustrates these two methods. In the first one, the most recent k samples are given equal weight, and all the previous data are forgotten; in the second one, the weights of the older samples are decreased exponentially. Here, we use the second approach since it is not clear (to us) how to decide the size of window at run-time. Furthermore, the rectangular sliding window is less appealing since it requires remembering all past values that are in the window.

Instead of recursive least squares, we could also use incremental gradient descent [66] to update linear model parameters. In gradient descent, we update the linear model parameters in small steps until we achieve a point where the solution does not improve

any more. Let x is a vector of parameters, y is the target output (corresponding to hardware event counts and power measurements, respectively) and β are the parameter weights. The predicted linear model output is given by βx. In incremental (stochastic) gradient descent, we update model parameter weights for each new sample, $i+1$ using:

$$\beta_{i+1} = \beta_i + \eta(y_{i+1} - \beta_i x_{i+1})x_{i+1}$$

Here, η is the step size. Higher values of η makes the model forget the past more rapidly. The linear gradient descent is computationally cheaper than recursive least squares regression, however, it is not as robust. The local minimas, plateaus, and high steep changes may cause it to oscillate. In our power estimation design, since the model power measurements y has an error term involved, robustness is particularly important for us. Thus, we do not pursue this option anymore, and leave it as a future direction.

4.6 Related Work

This chapter presents a runtime, feedback-based full system energy estimation model for battery powered devices. The run-time system maps hardware and software counters to power consumption values using first order, linear regression equations. The most closely related work is on HPM-based, static, linear models for CPU and memory energy estimation [7, 38, 78, 39, 35, 12]. These have already been described in previous chapter.

One of the primary challenges in this study is to find an optimum set of parameters that explain power consumption, however, simulating each possible parameter combination is hard since there are a large number of factors that shape CPU power behavior. In [44], Lee et al. suggests an efficient and statically sound approach that reduces design space-size considerably. In their simulations, they demonstrate that their regression models can estimate CPU energy consumption with a 4.3% error rate. However, their model does not use any feedback and is evaluated only for a hypothetical CPU. Like all the other studies above it does not explore any full-system energy consumption challenges. Lee et al.'s approach is complementary to our work as it provides a way for our to design better regression models that uses feedback from battery management unit.

The static models above are undoubtedly useful in characterizing program energy consumption, however they are limited by the static workload that they are developed on. This study proposes a novel approach to dynamically model and estimate energy consumption on low-power embedded devices.

4.7 Summary

This chapter presents an adaptive, feedback-based energy estimation model for low-power embedded devices such as HP hand-held computers and Stargate sensor network

devices. The presented model estimates full system energy consumption of programs using hardware and software counters. The system starts with an initial model and gradually improves it using dynamic feedback from the battery monitoring unit within the device. We evaluate our model using a large set of applications, and discuss its stability in the presence of measurement errors. Our results show that the proposed model can predict energy consumption with 1.0% error rate for computational bound programs and 6.6% error rate for tasks that are both communication and computation bound.

Chapter 5

Predicting System Resources For Reducing Energy Consumption

Previous chapters discussed modeling and predicting energy consumption. They presented challenges, and proposed novel solutions. These solutions demonstrated that fine-grain energy cost determination of tasks is possible with a carefully designed model coupled with a battery monitoring unit. Future power aware operating systems and applications like [55, 65, 87, 86] can be coupled with these models to detect and optimize the tasks that are most energy expensive. These operating systems use different strategies to reduce energy consumption of tasks. The two most important strategies are computation offloading and dynamic voltage scaling, which we described in detail in Chapter 2.

Key to the efficacy of dynamic voltage scaling and computation offloading is the low cost of their use and accurate prediction of future application, workload, and resource behavior. Techniques that optimize energy use must estimate the demand and the supply

of resources to determine which optimization to apply and when. If the estimates are incorrect or the application of the optimization introduces significant overhead, the techniques may be unable to extend battery life or actually shorten it.

This chapter discusses predicting demand and supply of these resources and proposes a non-parametric, time-series approach to estimate their future values using their past behavior. The next section presents extant resource demand and supply prediction systems. Section 5.2 presents the high level ideas behind our proposed approach. Section 5.3 details the design. Section 5.4 validates the proposed model. Finally, Section 5.5 summarizes the chapter and concludes.

5.1 Extant Resource Prediction Systems

Prediction of resource availability and performance is a widely studied field of research. This section overviews representatives of common resource prediction methods that are employed by studies that target energy consumption. It focuses on techniques that are online, require no modification to the application, and that are executed *on the device* itself, for which the overhead of the approach is as important as the accuracy it achieves. This section describes each prediction strategy in the context of the particular resources (CPU, network bandwidth, etc.) for which they are used.

CPU Availability

Systems employ CPU availability prediction to estimate the CPU time a process or sub-process task consumes [19, 28]. These predictions are used by the execution environment to guide task scheduling and processor scaling decisions [77, 26, 60, 41, 19, 28]. A common technique for estimating CPU load is one that gathers load statistics via various operating system utilities and interfaces, such as `vmstat` and `top` in UNIX. CPU estimation techniques range from very simple to complex and thus vary in agility, overhead, and accuracy. Agility is the degree to which a prediction utility can react to and adjust for variance in measured, history data.

PAST scheduling [77] assumes that the CPU load in the next interval will be the same as the most recent CPU load measurement. This forecaster is very agile since it immediately responds to changes in CPU load. However, such a response can have a negative effect on accuracy when recent CPU spikes are outliers (*noise*) and short lived, i.e., not good estimates of future behavior.

To overcome such limitations, other CPU prediction techniques filter out noise with more sophisticated techniques. The Odyssey prediction system represents such systems. Odyssey estimates CPU availability by first assuming that CPU cycles are evenly distributed among all processes. It then uses an exponential decay technique (i.e., a

smoothing filter) to filter out noise. The Odyssey CPU prediction model is:

$$S_{cpu} = \frac{P}{N+1} \tag{5.1}$$

where P is the processor clock speed, N is the number of runnable processes and S_{cpu} is the available CPU cycles. Odyssey uses a smoothing filter to estimate the number of processes in the next interval:

$$N_{t+1} = \alpha N_t + (1-\alpha)n(p) \tag{5.2}$$

In this equation n is a function of observed number of processes in the current interval and defined as:

$$n(i) = \left\{ \begin{array}{ll} n_r - 1 & \text{If } p \text{ is runnable} \\ n_r & \text{Otherwise} \end{array} \right\}$$

where n_r is the number of runnable processes at time t.

The AVG_n policy [77] is another popular CPU prediction technique that directly decays the measured CPU load over the last k intervals. Since this policy is simply an extension to PAST policy, it inherits the same weaknesses (i.e. static, parameterized). AVG_n policy is less agile than PAST but it is more resilient to noise in the network. The BEST [3] policy estimates the CPU demand using a similar mechanism. It predicts the periods of multimedia tasks as a weighted average of their previous task periods. It truncates periods that are longer than a threshold value. BEST policy adds predicted period to current time to find the task deadline, and schedules the task that has earliest

deadline. There are also several other CPU load prediction techniques that are based on observation heuristics [26]. Such techniques, however, are less general since they have many parameterized heuristic rules designed to optimize performance on the particular workload that they are intended for.

A more recent study [72] indicates that a single, parameterized method may not be the best choice across different workloads. In their study, Sinha et al. compared four CPU load prediction techniques, including exponential smoothing, moving averaging, least mean squaring and a purely probabilistic technique called expected workload state, using three real workloads. Their results showed that least mean squaring was better than the others, in average. However, the best predictor varied from one workload to another [71].

Network Latency and Bandwidth

Many embedded systems employ prediction techniques for network latency and bandwidth. Two common uses of such techniques are task scheduling for distributed devices and computation offloading. Computation offloading is a technique in which the system executes processes or tasks on more capable or wall-powered computer systems to conserve the battery power or extend the capability of mobile, resource-constrained devices [20, 55, 40, 2].

To estimate network latency, extant prediction systems use passive observations of RPC packets to compute the round-trip time and throughput of network [55]. Since, network performance is highly variable, noise can severely degrade the accuracy of network bandwidth estimations. To improve accuracy, other prediction systems use an exponential smoothing filter much like that used above for CPU [55]:

$$new = \gamma(measured) + (1 - \gamma)old \tag{5.3}$$

The value of the exponential decay factor (γ) determines the agility of the method. A smaller value increases the responsiveness, but decreases the technique's ability to filter out noise. Thus the accuracy of the method is highly dependent on the choice of the parameter. Since network latency and bandwidth exhibit different performance characteristics, users must identify multiple parameterizations (γ) for the filter function of each. Moreover, for a single metric (latency or bandwidth), the filter requires different parameterizations for *different* network technologies to achieve the best accuracy.

To overcome the limitations of a large number of parameterizations and instability, researchers have developed a network performance estimator that implements two exponential smoothing functions in a single forecasting system [40], a so-called flip-flop predictor. The parameters used by this predictor are commonly at opposite ends of the spectrum to capture the benefits of both agility and smoothing. Both predictors execute concurrently, however, the estimator uses the one with the larger parameter (that enables

more smoothing) as long as the approximate standard deviation of the predicted value is in a predetermined range with respect to the smoothed mean. The estimator switches to the agile version otherwise. This design has an important advantage over previous models as it is more accurate and can adapt more effectively to dynamic changes in the system.

Power Consumption

In resource restricted systems, because of the lack of a mechanism that can measure power consumption (which we address in the previous two chapters), the energy consumption of a task is rarely used as a direct cost metric. One exception is Remote Processing Framework. The Remote Processing Framework (RPF) [64] predicts the energy consumption of future tasks as a function of their previous energy consumption cost. It uses this to determine whether a task should be executed locally on the device or remotely on a wall-powered server, i.e., whether computational offloading should be performed. RPF does not detail how to collect these measurements, however, it suggests that the Advanced Power Management (APM) interface of the operating system can be used. To estimate task power consumption RPF uses this smoothing filter:

$$f_{n+1} = (1 - \alpha) * \frac{\sum_{i=n-k}^{n} v_i}{k} + \alpha * f_n \tag{5.4}$$

where f_i is the forecasted value, v_i is the measured value and i is the measurement index. α and k determine how conservative the forecaster is: A small k combined with a large α will result in higher responsiveness to recent changes.

Note that the RPF smoothing filter (Equation 5.4) is the same as the equation for CPU prediction prediction (Equation 5.2) when $k = 1$. In addition, the smoothing filter is the same as the bandwidth and latency prediction function in Odyssey (Equation 5.3) when $k = 1$ and $\alpha = 1 - \gamma$.

Application CPU Demand

The CPU demand of an application is highly dependent on the nature of the application itself. However, when no application-specific information is available or its collection is infeasible, prediction systems can estimate CPU demand using application history logs. The prediction system described in [54] employs such a methodology. This methodology is popular and likely to be successful for embedded devices, since it does not require any effort by the user or application programmer, access to program source code, or no modification to the program.

In such systems, an online, learning predictor maintains program-specific coefficients that are used to model the CPU demand of the application for a particular input dataset. Computing the *initial values* of coefficients requires off-line training unfortunately. However, once the initial values are set, the system updates the coefficients

using recursive least squares regression with exponential decay (LSQ). Given the characteristics of exponential decay, LSQ gives more weight to the recent observations.

LSQ can efficiently predict the value y when it is dependent on a set of parameters x such that $y = Ax + w$, and w is the measurement error or noise. The general formula for recursive LSQ to estimate CPU load of tasks is:

$$A_k = A_{k-1} - P_k\{x_k x_k^T A_{k-1} - x_k y_k\}$$

$$P_k = \{P_{k-1} - P_{k-1} x_k [\alpha + x_k^T P_{k-1} x_k]^{-1} x_k^T P_{k-1}\}/\alpha$$

where α is the decay factor and y_k is the measurement at time k. In the equation above, y_{k+1} is predicted by $A_{k+1} x_k$. The P_k matrix is commonly referred to as the *history* or *filtering factor* [85].

This technique performs well for augmented reality applications – a popular application domain for mobile devices. Such programs render pictures as a camera scans a set of scenes. Since commonly scenes overlap, their transitions are smooth. That is, the resource consumption behavior for the generation of a scene is similar to that of a neighboring scene. As a result, the performance data varies smoothly from scene to scene, enabling a prediction system that uses exponential smoothing to produce accurate predictions of CPU demand. As mentioned previously, a limitation of recursive least mean squaring is that numerical computation errors can accumulate after each recursion causing algorithm to become unstable and diverge [8].

5.2 Proposed Non-parametric Resource Prediction Tool

All extant prediction methodologies require user-specified parameterizations to forecast the cost of various resources. Users must identify the appropriate parameters through empirical evaluation or using a complex, off-line learning process. Unfortunately, the parameters are specific not only to the executing application but also to individual tasks within an application. As a result, the parameterization may not work well across applications or even across the tasks of a single application. There are also methods that mitigate this problem by requiring more user- or application- feedback [18]. Moreover, existing systems use a number of different prediction strategies (each requiring training and parameterization) for different resource types (e.g. CPU, network performance, and power consumption).

We address the problem of resource prediction using a time series approach; that is we assume the resource demand and supply measurements taken over time have an internal relation –such as autocorrelation, exponential trend, etc., and we exploit this relation for predicting future behavior. Obviously, this assumption may not always hold true, or the relation between the time series data may not be that obvious to exploit. In these cases, using custom heuristics may be a better approach. In addition, it is sometimes better to model resource demand using data-dependent functions [53], however, this approach requires (sometimes substantial) developer effort.

A time series approach has significant advantages. Consider network bandwidth. The future availability of a network link depends on both the physical characteristics of the medium, and the machines that share and use it. Furthermore, the protocols that use the network have their own characteristics. It is hard to account for each such parameter in a model. In this case modeling network bandwidth as a time-series reduces complexity significantly.

Time series approaches have been used successfully, as the previous section shows. The extant time-series predictors are parametric and static. Our time-series approach is complemetely different. Specifically, it is one that is *non-parametric, automatic, adaptive, and agnostic of resource type and application behavior*. That is, it is a single system that makes accurate predictions of any resource type for any application – without requiring application modification or participation by users for parameterization and off-line training. Moreover, it is appropriate for resource-constrained, mobile, systems, i.e., it consumes few device resources to make accurate predictions. The system is called *NWSLite*.

5.3 Design Rationale

NWSLite is an extension of the Network Weather Service (NWS) [80], a freely available toolkit [56], originally developed for the *computational grid* [22, 6]. The

computational grid is a computing paradigm for the development of software systems that enables dynamic acquisition of resources from a heterogeneous and non-dedicated resource pool. Grid systems are high-performance, large-scale, distributed systems that require applications to adapt to the dynamically changing systems on which they are executed as well as to highly-variable resource performance. To extract performance from these systems, application schedulers must use predictions of future resource behavior to determine how the application can best use the available resources.

The NWS operates a distributed set of performance sensors, from which it periodically, and unobtrusively, collects performance measurements. The sensors apply a set of statistical forecasting techniques to individual performance histories and generate forecast reports for the resources being monitored. The NWS disseminates these reports via a number of different APIs in near-real-time [81]. Currently, the NWS provides sensors for end-to-end TCP/IP bandwidth and latency, available CPU and memory, battery power, and disk storage, and is used in a large number of different Grid technologies.

NWS prediction uses a mixture-of-experts approach to prediction instead of relying on a single model. It implements a large set of models, each having its own parameterization. Given a performance history of observed measurement values, it generates a forecast for each measurement. NWS ranks each predictor by computing the prediction errors (the difference between measured and forecasted values). Each time a forecast is requested, NWS recalculates the ranking across all predictors using the most recent

	Name	Average Cost
1	**Last Value**	0
2	**Running Mean**	3
3	**5% Exp Smooth**	3
4	10% Exp Smooth	3
5	15% Exp Smooth	3
6	**20% Exp Smooth**	3
7	30% Exp Smooth	3
8	40% Exp Smooth	3
9	50% Exp Smooth	3
10	75% Exp Smooth	3
11	90% Exp Smooth	3
12	5% Exp Smooth, with 0.1% trend	10
13	10% Exp Smooth, with 0.1% trend	10
14	15% Exp Smooth, with 0.1% trend	10
15	20% Exp Smooth, with 0.1% trend	10
16	30% Exp Smooth, with 0.1% trend	10
17	Median Window 31	88
18	**Median Window 5**	16
19	Sliding Median Window 31	124
20	Sliding Median Window 5	26
21	30% Trimmed Median Window 31	106
22	30% Trimmed Median Window 51	169
23	Adaptive Median Window 5-21	171
24	Adaptive Median Window 21-51	455

Table 5.1: NWS forecasters and the approximate costs of each. The cost in column three is given in units of floating point operations performed.

history and chooses the most-accurate model. The implementation of NWS that we extended uses the 24 prediction models shown in Table 5.1.

This mixture-of experts method achieves its accuracy by employing wide range of statistical models, each of which may be most appropriate at a given time, for a given resource. This method also has other important advantages. First, even though the individual NWS models may be parametric, the overall system is not. The only input to the system is the measurement history, i.e., the NWS is agnostic of the resource to which the measurement belongs. Second, NWS can easily adjust itself to changes in the characteristics of the data series by switching to another model. Third, it can be used on any type of data for which measurements can be made. There is no distinction between CPU availability and network bandwidth, for example.

Because the NWS was originally designed to support high-performance applications in wired settings, its designers put a premium on speed and extensibility. As such, it consumes significant resources to perform a single prediction since many models are evaluated at once. The *Average Cost* column shows the number of floating point instructions executed for each predictor (all are computed for each forecast made) on average. To enable its use in resource-restricted environments, we have significantly reduced this consumption without sacrificing appreciable accuracy. Here, we use dynamic floating point instructions as the cost metric because of their high cost.

Given a history of measurements and their predicted values, a common way of measuring the prediction error is using the square of the errors:

$$E = \sum_{i=1}^{n} (f_i - v_i)^2 \tag{5.5}$$

where f_i is the output of the predictor, v_i is the measurement and n is the length of history.

Since the NWS uses a mixture-of-experts approach, all forecasters are invoked logically in parallel and a single winner is selected and used for the next estimation. We use zero-one integer variables $s_{i,j}$ to denote the winning forecaster:

$$s_{i,j} = \left\{ \begin{array}{ll} 1 & \text{If model } j \text{ is used to predict} \\ & \text{measurement } i \\ 0 & \text{Otherwise} \end{array} \right\}$$

Specifically, if $s_{i,j}$ is 1, the ith forecast is made using predictor j. If $s_{i,j}$ is 0, the predictor is not the winner for the ith forecast. Let k be the number of models in NWS, using (5.5), the prediction error of NWS is:

$$E = \sum_{i=1}^{n} \sum_{j=1}^{k} (f_i - v_i)^2 s_{i,j} \tag{5.6}$$

Similarly, it is possible to compute the cost of using the winning forecasters (in terms of floating point instructions, c) as:

$$C = \sum_{i=1}^{n} \sum_{j=1}^{k} c_j s_{i,j} \tag{5.7}$$

Theoretically, NWS can be optimized by running it with a different combination of internal models on a set of representative data and then removing the least efficient ones. However, the search space is prohibitive: There are a total of 2^{24} combinations. To reduce the search space, we use a heuristic that evaluates how much the total computation cost and error would change if one substitutes a forecaster u with another forecaster v throughout the series.

The formal definition of this process is:

$$s'_{i,j} = \left\{ \begin{array}{ll} 1 & \text{If model } j \text{ is winner forecaster} \\ & \text{for measurement } i \text{ and } j \neq u \\ 1 & \text{if model } j \text{ is not winner forecaster} \\ & \text{for measurement } i \text{ and } j = v \\ 0 & \text{Otherwise} \end{array} \right\}$$

where $E_{u,v}$ and $C_{u,v}$ are Equations (5.6) and (5.7), using $s'_{i,j}$ instead of $s_{i,j}$.

We employ real measurement data (i.e. performance traces) to empirically evaluate the overhead and accuracy of each NWS predictor from various embedded system resources: Wireless and wired network bandwidth and latency, CPU load, and task CPU demand. We compute $E_{u,v}$ and $C_{u,v}$ for every pair of u and v using a set of six rep-

	1	2	3	4	22	23	24
1	0	0	0	0	0	0	0
2	0.138	0	0.001	0.001	0.011	0.01	0.048
3	0.283	0.069	0	0	0.034	0.039	0.095
4	0.267	0.084	0	0	0.026	0.033	0.087
22	0.004	0.001	0	0	0	0	0.001
23	0.003	0.004	0	0	0	0	0
24	0	0	0	0	0	0	0

Figure 5.1: Error matrix for a real input. The matrix shows the change in error in percentages when forecaster v (in rows) is substituted with forecaster u (in columns). For example, substituting forecaster 2 (Running Mean) instead of 23 (Adaptive Median Window-51) increases error rate only 0.1%.

resentative traces and record the results in a matrix with u as the rows and v as the columns.

This representation provides a very compact form to evaluate the efficiency of each model: Every column of the matrix shows how much the error rate changes if one uses v instead of u. For example, $E_{2,1}$ shows the new error if *last value* takes place of *running mean*. Thus, if the $E_{2,1}$ is smaller than original NWS's error rate for *all* the trace files, then we consider *last value* to be a better predictor than *running mean*. Similarly, if in an extreme case, all the values of column 2 are smaller than original NWS's error rate, then the *running mean* outperforms the original NWS. Even though, this is theoretically possible, we did not come across an example of such a case.

	1	2	3	4	22	23	24
1	0	0	0	0	1.39	3.73	0.32
2	−0.02	0	0	0	3.05	8.19	0.71
3	−0.05	0	0	0	3.43	9.24	0.8
4	−0.06	0	0	0	0.12	0.33	0.03
22	−0.06	−0.06	−0.06	−0.06	0	0.1	−0.05
23	−0.18	−0.17	−0.17	−0.17	−0.11	0	−0.16
24	0	0	0	0	0	0	0

Figure 5.2: Cost matrix for a real input. The matrix shows the change in cost in percentages when forecaster v (in rows) is substituted with forecaster u (in columns). For example, substituting forecaster 2 (Running Mean) instead of 23 (Adaptive Median Window-51) provides a 6% decrease in cost.

Figure 5.1 and 5.2, show the error and cost matrices for an example dataset. The numbers to the leftmost and topmost of the matrices show the enumeration of forecasters (given in Table 5.1). The numbers in the matrices show the change in error and cost in percentages after substituting a forecaster u (in rows) with a forecaster v (in columns). For example, substituting Adaptive Median Window-51 (number 23-shaded row) with Running Mean (number 2-shaded column) increases the error by 0.1% and decreases the cost by 6%.

Given the evaluation matrices, we employ a set of empirical rules to eliminate forecasters. Basically, we remove any model

- that has more than 1% error rate across all traces,

- for which there is another model with significantly lower cost that can replace it with a small increase in error ($< 5\%$), and
- for which there is a combination of other models that enable a similar error rate.

Rule 1 eliminates many models immediately. For example, replacing *30% trimmed median window 31* with *running mean* results in an error increase by at most 0.2%. On the other hand, for *median window 31*, replacing it with the *running mean* increases error rate less than 0.2% in 5 of the 6 traces. In the remaining trace, *median window 5* produces the same error rate as *median window 31*. As such, we include *running mean* and *median window 5* and omit *30% trimmed median window 31*. Using this process iteratively, we identify five predictors (shown in bold in Table 5.1). These techniques trade off cost and prediction error most effectively.

This methodology for discovering the five NWSLite forecasters is similar to off-line, profile-based optimization for which researchers use one set of program inputs to collect profile information and to guide optimization, and a different set of inputs to evaluate the performance of their approach [43, 74, 42]. We use six traces to identify NWS forecasters that enable high accuracy at low cost. We evaluate NWSLite using over 300 traces that are different from these six.

Name	Trace Size	Description
Application	20 traces 17870 predictions	Interactive, 3-D rendering application CPU demand. Measurements are CPU time from user request to program response.
Network Bandwidth [56]	132 traces 750476 predictions	Observations of 64Kb-1Mbyte TCP data transfers. 3 configurations: UIUC LAN (inter-cluster), UIUC campus-wide network (intra-cluster), and cross-country Internet (UIUC-UCSD)
CPU load [56]	59 traces 6000697 predictions	Fraction of CPU occupancy time a standard user process can obtain. Observations are in 10 seconds intervals.
Network Latency [56]	134 traces 750305 predictions	Round trip time of TCP. Transferring 4 bytes and measuring acknowledge time. Granularity levels same as network bandwidth.
Wireless Bandwidth [70]	1 trace 3028 predictions	4 access points on same subnet. Traces include 195 users, 300000 flows and 4.6 GB of network traffic. Bandwidth measured in 1 minute intervals.

Table 5.2: Datasets used for evaluation

5.4 Validation

To empirically evaluate the efficacy of NWSLite, we performed experiments using a wide range of datasets, applications, and metrics. The following subsection describe the experimental methodology (datasets and applications). Section 5.4.2 details the metrics, and Section 5.4.3 presents the results.

5.4.1 Experimental methodology

To empirically compare the resource forecasting system NWSLite to extant approaches to resource performance forecasters, we collected traces from a wide range of resource types: CPU demand (execution time) of application tasks, wired and wireless network bandwidth, wired network latency, and CPU availability. We then used the NWSLite and competitive approaches to make predictions using the trace data. In total, we performed experiments on 346 traces which produced more than 7 million predictions. All of the traces, with the exception of application execution times, were made freely available via web-sites of research groups around the country [56, 1, 27]. We provide the details on the different datasets in Table 5.2 and we refer to each of the different types of data sets (application execution times, CPU availability, bandwidth, latency, etc.) as "groups".

We generated execution time traces, i.e., CPU demand, using the 3-D rendering applications used in similar studies [54, 53]. The applications and inputs are shown in Table 5.3.

GLVU [25] allows navigating inside a 3-D scene by rendering the scene from any viewpoint of user. From an augmented reality view, Radiator [79] complements GLVU by computing the lighting effects for a given scene. Both applications can easily be divided into *operations* [54], which is a suitable unit for remote execution and fidelity adjustment. An operation (which this chapter also refers to as a task) is the smallest

		Applications	
Input Scene	Scene Size (bytes)	GLVU	Radiosity
castle	385391	Yes	
cessna	200553	Yes	Yes
chevy	678806	Yes	
cloister	7816848	Yes	
cup	97113		Yes
dragon	3382396		Yes
ground-table-land	640939	Yes	Yes
ground-riverain-valley	634007		Yes
shuttle	15658	Yes	Yes
venus	3483433		Yes

Table 5.3: NWSLite evaluation benchmarks. We collected 10 trace files per application (3-D scene rendering programs) using different inputs and navigation paths. Empty entries indicate that the application failed to process the particular scene; ”Yes” entries are those inputs we employed for this study. We processed some inputs multiple times (to total 10) using different navigation paths.

user-visible execution unit, such as viewpoint change in a rendering operation. For each application we rendered a set of 10 scenes which produced a total of 17870 operations. Table 5.3 shows these inputs. Since some scenes are not compatible with both applications, we used some inputs multiple times using different navigation paths. We consider the prediction performance for applications to be the accuracy with which the prediction system forecasts the CPU demand of each task.

The bandwidth, CPU availability, and latency data were collected as a part of the NWS project [56]. NWS network sensors use active network probes to collect TCP/IP latency and bandwidth data on a group of geographically distributed hosts connected via local, wide area, and Internet networks. Each probe establishes a TCP connection,

transmits a fixed amount of data, and tears down the connection. Network sensors measure network bandwidth using a 64 KByte data transfer and network latency using a 4 byte data transfer.

The NWS CPU sensors combine the information from Unix system utilities *vmstat* and *uptime* with periodic active CPU occupancy tests to provide measurements of CPU availability. The *uptime* utility reports the average number of processes in the run queue over the last one, five and fifteen minutes. The sensor uses the average load over the one minute period and computes the CPU availability by using the idle, user, and system time output from *vmstat* utility. The CPU availability is measured as the fraction of CPU occupancy time a standard user process can obtain.

The wireless bandwidth traces were collected during the SIGCOMM'01 conference and made public in [70]. The conference building was covered with four 802.11b access points. The traces span a 3 day period capturing 300000 flows generated by 195 users consuming a total of 4.6 GB of bandwidth.

5.4.2 Evaluation Metrics

This section evaluates NWSLite and its competitors in terms of both accuracy and computational cost. It uses three metrics to evaluate predictor accuracy and instruction count (both total and floating point) to evaluate predictor cost.

The first of the three evaluation metrics is **error deviation**:

$$MSE = \frac{\sum_{i=1}^{n}(x_i - y_i)^2}{n}$$

$$\text{Error deviation} = \sqrt{MSE} \tag{5.8}$$

where x is the set of n predictions and y is the set of n corresponding observations. The mean square error (MSE) is the average square prediction error over the n pairs, (x,y). The error deviation is the square root of the mean square error. Error deviation describes the error in absolute terms and represents (in analogy) the *standard deviation* of the errors with respect to the *expectation* constituted by the forecast. Error deviation accounts for outliers and is more sensitive to incorrect predictions than is *absolute error* in which the absolute value of the error is used.

However, the error deviation is most meaningful when comparing the performance of predictors on the same time series. To provide a comparison across different series, we use the ratio of error deviation over the average observed value, i.e., the **relative error rate**:

$$\text{Relative error rate} = \frac{\sqrt{MSE}}{observed_mean} \tag{5.9}$$

This metric provides insight into how severe the error is in terms of the magnitude of the average measured value. For example, an error of 2Mb/s is large in a 10Mb/s link, but may not be significant in a 100Mb/s link.

The third metric is similar to relative error rate, however, instead of using the mean as the expected value, it uses the absolute value of the forecast. This metric, called **predictability**, indicates how predictable the series is relative to the forecasts it generates. It differs from the *relative error* in that it treats each forecast as a *conditional expectation* that it uses to normalize the error, instead of using the overall measurement mean. Its definition is:

$$\frac{\sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i|}}{n} \tag{5.10}$$

5.4.3 Predictor Accuracy

This section presents the results from empirical comparison between NWSLite and competing prediction systems: The Network Weather Service (NWS), Odyssey (LSQ and ODY-BW,LAT), and the Remote Processing Framework (RPF). We implemented all of forecasters as efficiently as possible using the C language; we compiled each using gcc and -O2 optimization. Unlike NWSLite and the NWS, the LSQ and RPF methods are parametric models and hence, require parameterization. For each model, we created a pool of parameter settings, that included the published values [54, 20, 55] as well as our own values, resulting in 18 different forecasters. For conciseness, we selected the best performing parameterization for each over all of the datasets we considered.

Description	Units	Avg	NWSLite
APP1 - best		148845.000	5287.856
APP2 - median	msecs	9179.390	1322.139
APP3 - worst		169753.000	135125.056
BW1 - within cluster		65.801	17.161
BW2 - cross-cluster	Mbits/sec	76.522	13.308
BW3 - cross-country		4.536	0.878
CPU1 - best		1.992	0.016
CPU2 - median	CPU	0.543	0.017
CPU3 - worst	fraction	1.391	2.672
LAT1 - within cluster		13.936	16.873
LAT2 - cross-cluster	msecs	2.345	8.309
LAT3 - cross-country		77.217	14.295
WBW	Kbits/sec	206.674	193.782

(a) MSE of NWSLite across benchmarks

Description	NWS	LSQ	RPF
APP1 - best	5358.179	8180.561	22013.694
APP2 - median	1329.372	2385.072	5702.085
APP3 - worst	138064.335	145384.166	186430.176
BW1 - within cluster	16.958	52.112	17.191
BW2 - cross-cluster	13.329	59.279	13.507
BW3 - cross-country	0.859	78.063	1.164
CPU1 - best	0.016	13.905	0.029
CPU2 - median	0.017	14.451	0.049
CPU3 - worst	2.684	3.113	2.661
LAT1 - within cluster	16.890	41.121	17.048
LAT2 - cross-cluster	8.319	46.829	8.337
LAT3 - cross-country	12.753	81.820	13.149
WBW	194.498	255.254	261.744

(b) MSE of competitors across benchmarks

Table 5.4: Error deviation for a set of representative traces. The third column of (a) is the average of the measured values. The remaining columns show the error deviation for predictors. The APP and CPU datasets are sorted with respect to ***error deviation / average*** and best, median and worst cases are shown. For the BW and LAT datasets, the average error deviation within cluster, across cluster and across country are reported.

Table 5.4 compares the **error deviation** (Equation 5.8) of the predictors using three representative traces, for brevity. In the application (APP) and CPU availability (CPU) datasets, we sorted the traces with respect to the *error deviation / average* of NWSLite and selected the best, worst, and median, which we report in the table. For the wired network data (bandwidth (BW) and latency (LAT)), we instead report data for three different types of links: intra-cluster, inter-cluster, and inter-campus (across country). For wireless (WBW), we only have a single trace and thus show data only for it.

The first three columns of the table shows the description, trace name, and value units for each trace. The third column, Avg, shows the average observed value. The final four columns show the error deviation for each of the four predictors: NWSLite, NWS, LSQ, and RPF. LSQ and RPF are parameterized as described in Section 5.1, and identify the best-performing, converging parameterizations of each technique.

The NWS and NWSLite have almost identical error deviations in every case. LSQ performs well for applications (as was shown in prior work [54]), but it is the worst-performing predictor for all other types of data. NWSLite performs better than LSQ and RPF in almost every case, and is significantly better than both LSQ and RPF in most cases. For example, in the application group, for both *shuttle* and *cloister* NWSLite performs 3 times better than RPF. The wireless dataset is especially challenging. All the forecasters show a high error rate.

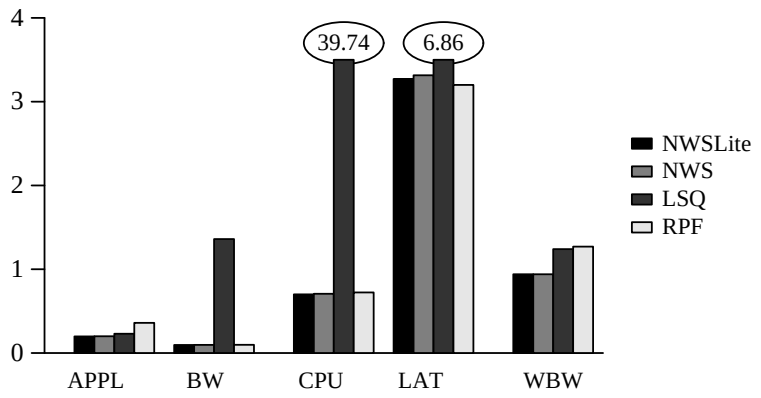


Figure 5.3: NWSLite relative error rate (Equation 5.9). This metric shows how severe the error is with respect to the average measured value. The LAT has the highest relative error rate among all forecasters, however, as most latency observations are very small (around 1 msecs), the absolute error is small.

Figure 5.3 shows the **relative error rate** of the predictors across all of the traces in each group. The information in the graph confirms the results of Table 5.4. NWSLite performance is very similar to that of the NWS; in all groups it enables the best prediction error. LSQ is ineffective for the bandwidth, CPU, and latency groups. RPF performs quite well for the CPU and bandwidth groups; and exceeds NWSLite performance for network latency by 1.5%. RPF is the worst predictor however, for the application and wireless groups. For the application group, the average error rate of RPF is 86% higher than that of NWSLite.

We also compared the performance of predictors with Odyssey's specialized smoothing filters for bandwidth and latency, which we refer to as ODY-BW and ODY-LAT

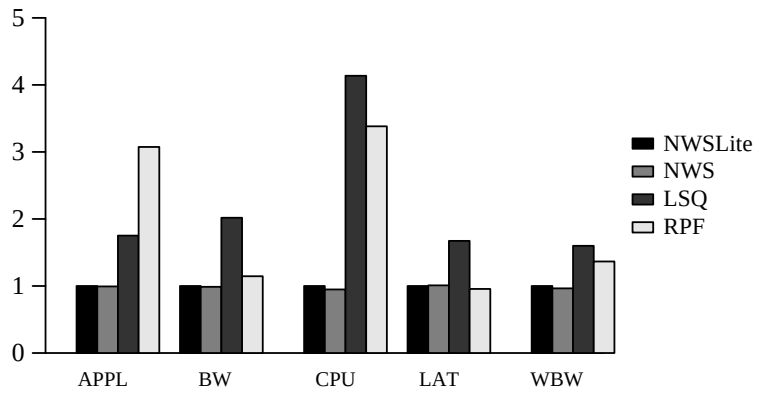


Figure 5.4: Predictor predictability (Equation 5.10). Due to high variation among forecasters, the values are normalized to NWSLite for each group. The lower the value, the more accurate the forecaster.

(omitted for clarity). ODY-BW performed 25% worse than NWSLite and ODY-LAT performed 19% worse than NWSLite.

Figure 5.4 shows the **predictability** (Equation 5.10) of the series given each predictor. This metric assumes that predictor is a valid conditional expectation that can be used to normalize the error at each point of the trace. The lower the value the more accurate the forecaster. Since the variance of the results is high, we normalized the results to NWSLite for each group.

The predictability results support the previous findings in Figure 5.3. NWSLite is as accurate as NWS in all cases, and it performs significantly better than the parameterized forecasters in most cases. The single exception is the latency dataset, in which RPF is the winner. However, the difference between RPF and NWSLite is very small. In contrast, the accuracy of RPF is significantly worse than NWSLite for the applica-

tion, CPU, and wireless bandwidth data, emphasizing the difficulty of finding a good parameterization for the general case. These results also show that, with the exception of the application dataset, LSQ always performs worse than the predictors based on smoothing-filters. In the application dataset, LSQ is approximately 40% more accurate than RPF, however, it is still significantly worse than NWSLite. The predictability of NWSLite is considerably higher than even the highly tuned predictors ODY-LAT and ODY-BW (not shown in figure). For the latency dataset, ODY-LAT is 13% less predictable than NWSLite; whereas in bandwidth dataset, NWSLite does 21% better than ODY-BW.

An interesting case is the behavior of RPF in Figures 5.3 and 5.4; even though the relative error rate of RPF is small, its predictability is not. This is due to the characteristics of CPU dataset - the CPU availability values are in the range $(0, 1)$, or $(0, n)$ if there are n processors. As such, most of the time the values are a fraction of 1. This results in a small value for the sum of square errors even though the errors are high relative to the expected value.

5.4.4 Computational Cost Of Prediction

In addition to studying prediction error, we also considered the cost of performing prediction on a resource-restricted device. Prior studies that use prediction on mobile devices do not consider the resource consumption of the predictors themselves, how-

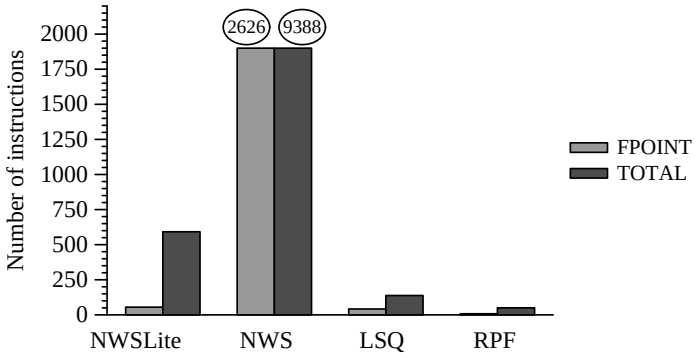


Figure 5.5: Forecaster cost as number of instructions executed, (floating-point (FPOINT) and TOTAL) per Prediction

Prediction System	Floating Point	Total Instructions	Execution time (microsecs)
NWSLite	55	592	381.34
NWS	2626	9388	10231.31
LSQ	42	138	295.27
RPF	8	50	154.9

Table 5.5: Execution cost comparison per prediction

ever, this cost may be significant especially on processors without any floating point co-processors.

Here, we first compare the predictors in terms of instructions required for one prediction. We extracted this information by using the SimpleScalar [10] simulator. Figure 5.5 shows the average cost of each predictor. NWSLite uses 55 floating point instructions per forecast. Even though this is more than the cost of RPF and LSQ, which use 8 and 42, respectively, the accuracy of NWSLite exceeds both of these predictors significantly.

As most resource-restricted devices lack a floating point co-pro-cessor, floating point instructions are very expensive. We break down the instruction counts into floating-point and non-floating-point instructions in the first two columns of Table 5.5.

We also executed the predictors on a real resource-restricted device: An iPAQ H3800 hand-held computer from Compaq [30]. The iPAQ has a 206 MHz Intel StrongArm CPU and runs Familiar Linux version 0.5.3. The execution times (in microseconds) are shown in the final column of the table. These times include the cost of IO to read the trace file from flash memory and to print the results.

The execution time of NWSLite is approximately 4% that of NWS but enables prediction accuracy that is nearly equivalent. Given that it requires only 381 microseconds to execute a prediction, including the IO, NWSLite is a more attractive solution for online forecasting using resource-restricted devices, than the parametric and less accurate models of Odyssey and the RPF.

5.4.5 Validation Summary

Table 5.6 summarizes the results of this chapter. To make the results comparable to prior studies [54], it reports summary performance in terms of percentile error. Here, the X percentile error, E_X, is the maximum *absolute* error for X% of the experiments. For example, for the bandwidth dataset, E_{95} of NWSLite is 25.6 meaning that 95% of the time the prediction error of NWSLite is within 25.6 kilobits/second. The reason

	E95				
	APP	BW	CPU	LAT	WBW
NWSLite	7336.000	25.699	0.043	24.566	351.090
NWS	7459.000	25.580	0.038	24.502	358.798
LSQ	13305.338	28.459	0.115	26.867	422.977
RPF	38696.700	25.561	0.209	24.915	533.047
ODY-LAT	8806.945	39.717	0.094	29.848	335.172
ODY-BW	7894.141	42.541	0.079	31.494	354.560

(a) 95 percentile error

	E90				
	APP	BW	CPU	LAT	WBW
NWSLite	3319.000	10.271	0.019	15.772	198.130
NWS	3343.000	9.601	0.018	15.801	202.771
LSQ	5866.552	14.105	0.058	16.415	230.591
RPF	17147.400	10.596	0.080	16.187	326.340
ODY-LAT	3759.839	9.923	0.025	16.318	197.429
ODY-BW	3458.320	7.384	0.021	16.883	192.992

(b) 90 percentile error

Table 5.6: Results in summary: Percentile Error. We define the X percentile error, E_X, as the maximum absolute error for X% of the experiments. The table compares the E_{90} and E_{95} of all forecasters for all 5 datasets and prediction systems studied.

we use absolute rather than relative error is to avoid skewed data in CPU and latency datasets. we report the results for NWS, NWSLite, LSQ, RPF as well as for the two other smoothing filters that was included, ODY-LAT (the Odyssey network latency predictor) and ODY-BW (the Odyssey network bandwidth predictor).

The results show that NWS and NWSLite are general enough that they perform well in all datasets. Even though parameterized forecasters can match NWSLite in some datasets, they fail in others. As an example, the performance of ODY-BW is close to NWSLite in APP dataset, but it is significantly higher in BW, CPU and LAT datasets. The same pattern also exists for ODY-LAT and RPF. RPF matches NWSLite in BW and LAT, but it is significantly worse in APP and CPU datasets.

Another pattern in the results is that both NWS and NWSLite perform better than all others when a higher percentage of predictions considered. This suggests that, NWS and NWSLite can better adjust themselves to sudden changes in performance patterns by switching to another model; the other models must simply rely on their static parameters.

The wireless bandwidth dataset is significantly different than other datasets. The error rates are very high, i.e., E_{90} is around 200Kbits/sec on a 11Mbits/sec link, hence none of the forecasters performed at a satisfactory level. This emphasizes the need for additional study of novel forecasters for wireless network bandwidth data.

The success of NWSLite results from its capability to dynamically switch between a carefully chosen set of competing models based on previously observed accuracy. If the dynamics of the observed dataset changes over time, NWSLite can adapt to the new conditions; the prediction systems of Odyssey and RPF cannot as such are data (input) dependent. For example, exponential smoothing with a gain of 0.05 can be the most accurate predictor at some point, however, a transient or permanent change can occur so that the running mean can become the most accurate. In this case, NWSLite will respond by switching to running mean if the change is persistent enough to cause the aggregate error ranking to change. Odyssey and RPF are statically configured by a set of pre-determined parameters. Thus, even though there are individual cases that other predictors can match the accuracy of NWSLite, they are unable to do well across dynamically changing series and to different types resource performance data.

The flip-flop filter extension to Odyssey [40], described in Section 5.1, incorporates some adaptivity by using two different parameter settings in its exponential smoothing predictor. However, exponential smoothing cannot always produce the best prediction accuracy (given any gain parameters). NWSLite incorporates exponential smoothing using two different gain factors but is more general and adaptive than this filter since it considers a wide range of other prediction techniques that can enable significant improvements in accuracy at low computational cost.

5.5 Summary

This chapter presents a light-weight, computationally efficient, prediction utility for mobile devices called NWSLite. NWSLite is an extension of the Network Weather Service (NWS), a dynamic measurement and forecasting toolkit designed and developed for adaptive application scheduling in Computational Grid environments (performance-oriented distributed systems). We identify 5 of the 24 NWS forecasters for NWSLite implementation, that trade-off computational cost for predictor accuracy most effectively.

We evaluate NWSLite using over 300 different traces of application execution times, CPU availability, wired network bandwidth and latency, and wireless bandwidth. In addition, we compare NWSLite to the NWS and to two other extant remote execution prediction systems. We find that NWSLite consistently outperforms the latter and achieves prediction accuracy similar to that of the NWS. However, NWSLite achieves this level of accuracy at a significantly lower execution cost than the NWS.

We show the utilization of NWSLite on a computation offloading platform, by evaluating it for resource supply and demand prediction using two computation offloading scenarios. In the first scenario, NWSLite beats two popular predictors, RPF and LSQ, by 67% and 14% fewer wrong decisions, consecutively. In the second scenario,

NWSLite beats those predictors even with a higher margin: 95% and 73% fewer wrong decisions. NWSLite achieves this rate without any significant increase in cost.

Chapter 6

Improving Computation Offloading

NWSLite can adapt itself to the changes in data conditions easily by using the dynamic predictor selection mechanism that it employs. Our findings, which we gather using statistically sound metrics, show that NWSLite outperforms its competitors significantly on a wide range of datasets. However, a key question that is remaining is how much this improvement in accuracy translates into actual energy savings. Here, we answer this question by evaluating NWSLite in a computation offloading setting.

The next section discusses the components required for a remote execution system and detail the significant parameters. Section 6.2 describes our computation offloading setting. Section 6.3 discusses the results. Finally, Section 6.4 summarizes the chapter and concludes.

6.1 Resource Prediction in Remote Execution

Offloading computation to remote, wall-powered, resource-rich servers can provide significant power savings. Computation offloading has to compute the cost of local execution and remote execution before offloading the computation to a remote server. To utilize the power consumption as efficiently as possible, the system should offload computation only when remote execution is expected to use less energy than local execution. In order to achieve this, the cost model must consider both the task execution characteristics as well as the highly-variable performance of the underlying resources that dictate computation and communication performance. Predicting the future state of these resources and application demand requires a high quality prediction mechanism. Chapter 2.1 describes remote execution in more detail.

Since the scope of this chapter is to compare NWSLite to other predictors, we employ a general cost model that assumes no I/O overlapping. We compute the available CPU cycles using the Odyssey model, which we give in Equation 5.1. We compute the local and remote execution cost as:

$$L_l = \frac{D_{cpu}}{S_{lcpu}} \tag{6.1}$$

$$L_r = \frac{D_{tx}}{S_{tx}} + \frac{D_{cpu}}{S_{rcpu}} + \frac{D_{rx}}{S_{rx}} + D_{rtt}S_{rtt} \tag{6.2}$$

where L_l and L_r stand for local and remote execution latency, D_{cpu} is the number of CPU cycles that the application requires to complete the task and S_{lcpu} is the available

CPU cycles on local machine, averaged in a period of one second. The remote cost is the sum of four constituent operations:

1. The time required for network transfer given the size of the demand for network send, and any needed program code (D_{tx}) and given the available bandwidth (S_{tx}) between the device and server;

2. The execution time at the server given the average number of CPU cycles available at the server (S_{rcpu});

3. The time for transfer of results, e.g., data, status, rendered graphics, etc., back to the device given the available bandwidth between the server and device ($\frac{D_{rx}}{S_{rx}}$); and;

4. The time required for handshake to establish connection, given the number of packet exchanges between local and mobile device (D_{rtt}) and network latency (S_{rtt}).

Since (4) commonly consists of very short packet communication between the device and the server, the handshake operation is impacted by the latency in the network link between the client and server ($D_{rtt}S_{rtt}$). L_l and L_r can be enhanced to compute power, to integrate computation fidelity or battery lifetime into cost functions.

```
int likely_offload()

{ // returns 1 if remote execution chosen, 0 otherwise

        L_L = predict_local_latency();

        if (L_L < 50 milliseconds) {

                return 0;

        }

        L_R = predict_remote_latency();

        if (L_L > L_R) {

                return 1;

        } else {

                return 0;

        }

}
```

Figure 6.1: Pseudocode for Scenario1 Decision Manager

6.2 Methodology

To better understand how much the improved accuracy enabled by NWSLite matters to a real remote execution system, we constructed two scenarios and simulated those scenarios using the principles that we described previously. In our simulation, we limited the computation offloading scenarios to one mobile device and one remote ex-

ecution server. We assumed that the local device is an HP iPAQ H3800 and the remote device is an IBM T23 laptop. The former machine has a 206MHz StrongArm CPU, while the latter one uses a 1132MHz Pentium III. We assume that both machines are not executing any other task.

The scenarios simulate computation offloading systems that have different goals. In Scenario1, the goal is to provide optimal user-interactivity. Many mobile applications, such as augmented reality applications, and games are user-interactive by their design, making response time a critical design parameter. For such applications, computation offloading is a viable option not only to improve response time but also to improve functionality [54, 41]. In Scenario2, the goal is to reduce power consumption as much as possible and to extend battery life. Scenario2 does not consider execution performance (latency) in offloading decisions.

The simulator reads the measured values of each of the computation offloading parameters, – CPU demand, local and remote CPU supply, and network latency and bandwidth –, from a file and predicts their future values by running separate instances of predictors for each type of data. Once the system computes the future values, it calls the decision manager, which determines whether local (a return value of 0) or remote computation (a return value of 1) will be used. The simulator also computes what the “right” or “best” decision is, once it reads the actual values, and computes various statistics for our use in the evaluation.

Figure 6.1 shows the pseudocode for the Scenario1 decision engine. The engine, which exists on the mobile device, offloads the task to the remote machine if the forecasted local execution time is more than 50 milliseconds (ms) and the forecasted remote execution time is smaller than that of the forecasted local execution time. The tasks that are estimated to have an execution time less than 50 ms are never offloaded, since the human perception system can not recognize delays that are less than 50 ms [11]. Thus, Scenario1 favors local execution, when appropriate, to reduce the stress on shared resources, such as the network and remote server. We discuss the implications of this choice on predictor efficacy in the next section.

In Scenario2, the decision process estimates power consumption for both local and remote execution and chooses the location that leads to lower power consumption. Unlike Scenario1, this scenario does not favor either local or remote execution (i.e. it does not take execution latency into consideration). Furthermore, its power computation function assumes that the local CPU and the wireless interface are in the idle state during remote execution. We detail the computation of power consumption later in this section.

We simulated Scenario1 using GLVU and Scenario2 using Radiosity. As we explained in Section 5.4.1, both of these applications can be split into tasks that can be offloaded to a remote server or executed locally. To measure the task CPU demand (i.e. D_{cpu}), we captured the task execution times, in microseconds resolution, as a user

was navigating 3-D scenes on a dedicated IBM T23 Linux laptop. We then computed the demand, in CPU cycles, as $D_{cpu} = t \times f$ where f is the CPU clock speed of the machine and t is the task execution time. Even though D_{cpu} is not completely accurate and portable across architectures due to differences in cache sizes and other CPU parameters (i.e. lack of floating point coprocessor in StrongArm), we ignore such discrepancies as our focus is the accuracy of predictors, not the efficiency of computation offloading.

In our simulations, we assume that only the input data is transmitted across the network. This is similar to prior approaches in [41, 64, 18]. Both GLVU and Radiosity tasks operate on an object file that contains the current scene. Since the size of this file is known beforehand, there is no need to predict network demand separately.

Prior to the data transfer, the client and the server has to initiate a session. In our model, the initial handshake, which includes a single message exchange, and the data transfer are done reliably, using the TCP protocol. Other implementations tend to be more complex [18] and use protocols like remote procedure call, however we do not discuss these for conciseness.

Each offloading decision requires prediction of four resources; network latency, network bandwidth, and local and remote CPU availability. Network latency is used to compute the cost of protocol handshake. Network bandwidth is needed to estimate cost of data transfer. CPU availability is used to compute the cost of local and remote

Parameter	Power (mWatts)	Description
p_{idle}	550	CPU idle- Wireless interface off
p_{busy}	1150	CPU highly busy - Wireless interface off
p_{tx}	2200	Data send over wireless
p_{rx}	2100	Data receive over wireless

Table 6.1: Power consumption of iPAQ under different scenarios [88]

computation. We use separate predictor instances on data histories to estimate next values of each of these resources.

We use Equations 6.1 and 6.2 to compute local and remote execution latencies in Scenario1. In Scenario2, to compute power consumption, we extend these equations such that:

$$C_l = \frac{D_{cpu}}{S_{lcpu}} p_{busy}$$

$$C_r = \frac{D_{tx}}{S_{tx}} p_{tx} + \frac{D_{cpu}}{S_{rcpu}} p_{idle} + \frac{D_{rx}}{S_{rx}} p_{rx} + D_{rtt} S_{rtt} p_{tx}$$

In the first equation above, C_l stands for local execution energy consumption. We compute it by multiplying local execution latency with p_{busy}, which is the average power consumption of a highly loaded handheld computer. Similar to L_r; C_r, the energy consumption during remote execution, is a sum of four factors:

1. The energy required for network transfer, which is network transfer time multiplied by p_{tx}, the average power consumption during wireless transmit;

2. The energy consumption while waiting for execution at the server, which is remote processing time multiplied by p_{idle}, the average power consumption in sleep state;

3. The energy required for network receive, which is network receive time multiplied by p_{rx}, the average power consumption during wireless receive; and;

4. The energy required for handshake to establish connection, which given the number of packet exchanges between local and mobile device is equal to network latency multiplied by (p_{tx}).

Table 6.1 gives the actual values of p, as measured by Li et al. [88] on real handheld devices.

We simulated each scenario using 3 input scenes. We chose the scenes arbitrarily, from Table 5.3, however, we were careful to choose one small, one medium and one large scene. For GLVU, we used Castle, Shuttle and Ground-Table-Land. For Radiosity, we used Cessna, Venus and Ground-Table-Land. We evaluated each scene using 32 different TCP bandwidth, network latency and CPU availability measurements that we describe in Section 5.4.1. We report the average results.

In each scenario, we compared the efficacy of NWSLite with RPF and LSQ using the best performing parameterization as we described in Section 5.4.3. We did not include NWS in our evaluations due to its high cost.

		Object Features		Number of	% of Offloading Decisions		
		Size (KB)	Complexity	Decisions	RPF	LSQ	NWSLite
Scenario1	Castle	385	Medium	78528	32.37	32.23	32.39
	Shuttle	15	Low	14080	64.32	66.82	65.00
	Ground-Table	640	High	26496	48.12	49.32	48.65
Scenario2	Cessna	200	Medium	12736	28.33	31.83	29.72
	Venus	3483	Very High	2720	9.63	12.21	16.58
	Ground-Table	640	High	5152	27.93	34.71	28.13

Table 6.2: Overview of 3-D objects.

6.3 Simulation Results

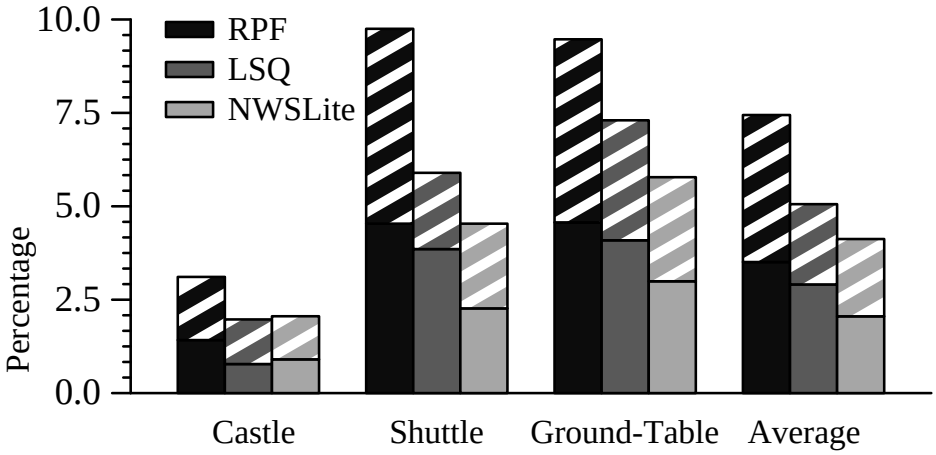
In this subsection, we evaluate how prediction effects the performance of the decision engine. There are two ways that the decision engine can fail for a given task: (1) the decision engine chooses local execution even though remote execution is more beneficial (2) the decision engine chooses remote execution even though local execution is more beneficial. We refer to the former as *wrong locals*, and the latter as *wrong remotes*. We use *wrong decisions* to refer to the sum of both wrong locals and wrong remotes.

Table 6.2 gives a brief overview of all the 3-D objects that we used. The first part of the table describes object features, including size, in Kbytes, and complexity, in scales that change from “low” to “very high”. A higher complexity object has more vertexes and edges per unit area, and more details such as 3-D information, and color. Such

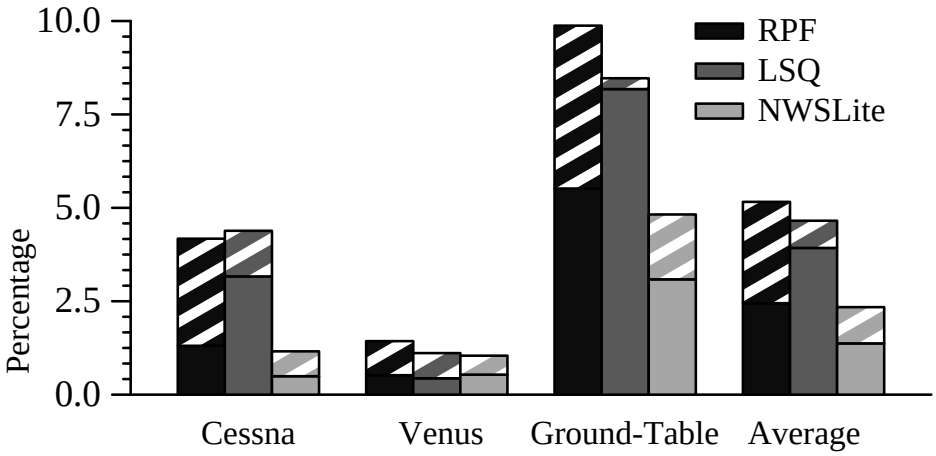
increase in complexity requires more network demand, but not necessarily more CPU demand, because rendering algorithms can intelligently prune out many details, such as the vertexes that are not visible, during processing. For example, even though Ground-Table is almost two times larger than Castle, its average rendering cost is approximately the same as that of the Castle.

The rest of the table gives the total number of task offload decisions and the percentage of offload decisions given by each predictor. A high number of task offload decisions shows that the user navigated the object for a longer duration, generating a larger number of tasks. This is typically the case for the GLVU tasks in Scenario1, since the tasks are shorter than the computationally demanding Radiosity tasks. The ratio of tasks that the predictors chose to offload varies from 10% (Venus in Scenario2) to 64% (Shuttle in Scenario1), depending on task characteristics. However, these numbers show only the degree to which predictors utilized local and remote execution, and does not indicate whether these decisions are correct.

In Figure 6.2, we compare each predictor in terms of wrong decisions. Each bar shows the percentage of wrong decisions. The striped and solid sections represent the wrong remotes and wrong locals consecutively. For example, for *Castle* approximately 2.6% of all decisions were wrong, and the ratio of wrong locals and wrong remotes were approximately equal. The last three bars show the average. We compute average by equally weighing all benchmarks; for example, if a scene has 450 wrong offloading



(a) Scenario 1



(b) Scenario 2

Figure 6.2: Percentage of wrong decisions. The striped and solid parts show wrong remote and local execution decisions, consecutively. NWSLite beats other predictors in each benchmark.

decisions among its 900 tasks, and another scene has 10 wrong offloading among 100 tasks, we compute the average of wrong offloading decisions as 30%, not 23%.

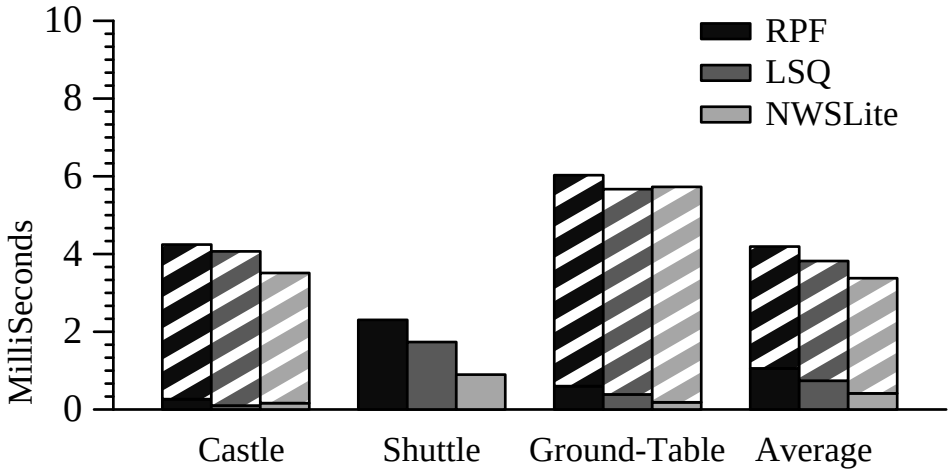
In a remote execution system, the computation offloading decisions show a boolean characteristic. Once the decision is wrong, it does not matter how close the predicted and the real value is. The possibility of a wrong decision increases when the gap between the cost of local and remote execution is small. That is because the small gap cannot compensate any prediction errors. An example is Venus in Scenario2. Venus is an extremely sophisticated scene. Due to its size, remote execution is very costly, however, local execution is not (i.e. even though Venus is approximately 5 times larger than Ground-Table-Land, the CPU demand is only 2.2 times larger in average). The large margin between the cost of local and remote execution compensates most prediction errors; therefore all the predictors can achieve very few wrong decision rates ($< 1\%$).

Overall, the wrong decision rate was less than 10% for all benchmarks. NWSLite was always better than the other predictors. In Scenario1, the the wrong decision rate for NWSLite, LSQ and RPF were 4.1%, 5.1% and 7.5%, consecutively. In Scenario2, NWSLite performed even better. The rate was 2.3% for NWSLite, and 4.7%, 5.2% for LSQ and RPF. This corresponds to 67% fewer wrong decisions than RPF and 14% fewer wrong decisions than LSQ for the first scenario. In the second scenario, the difference between NWSLite and other predictors is even larger; NWSLite gave 95% and 73% fewer wrong decisions than RPF and LSQ, consecutively.

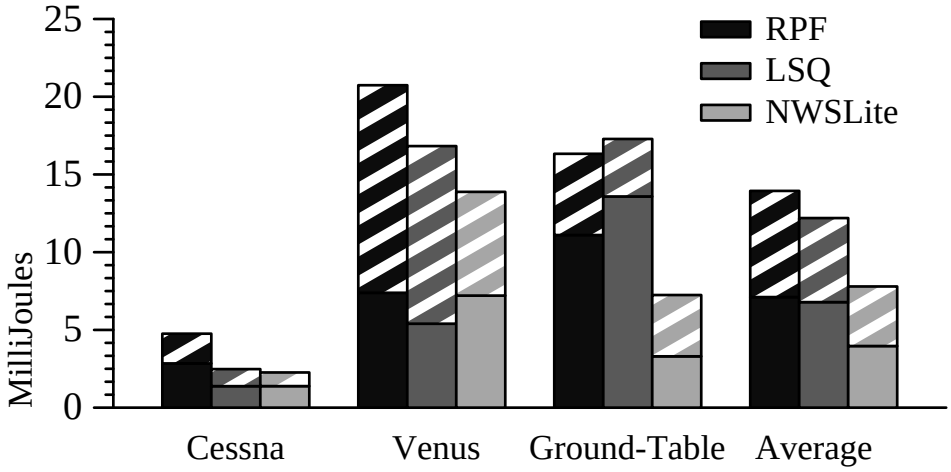
The wrong decisions were almost equally distributed among local and remote execution except the *Ground-Table-Land* benchmark in Scenario2. In Ground-Table-Land, only for the LSQ predictor, wrong locals were dominant. A plot of cost function unveils an interesting phenomenon: even though remote execution cost is stable, there are frequent, steep changes (i.e. dips) in local execution cost. When such a dip occurs, LSQ tends to over-correct its parameters resulting a steeper reduction (i.e. an underestimation) in local cost estimation, resulting in many wrong local execution decisions.

In Figure 6.3, we compare the cost of wrong decisions. We compute the cost as $\sum c_i/n$; where c_i is the cost of a **wrong decision** i, and n is the number of all tasks. We compute c_i as the amount of extra response time -or extra energy consumption, depending on the scenario- between the correct decision and the wrong decision. In other words, this metric gives the expected wrong decision cost per task. The results are in ms for Scenario1, and in millijoules (mJ) for Scenario2.

Figure 6.3 shows that, in Scenario1, there is a very uneven cost distribution among wrong locals and wrong remotes. In Castle and Ground-Table-Land, most cost is due to wrong remotes, in Shuttle **all** cost is due to wrong locals. This is due to the asymmetric nature of Scenario1; the computation offloading decision is given only when the task is expected to last more than 50 ms, therefore only large tasks are offloaded and a wrong decision adds a huge error. This effect is clear in Castle and Ground-Table-Land, but not in Shuttle. Due to the relatively lower CPU demand of Shuttle, (i.e. a very small



(a) Scenario 1



(b) Scenario 2

Figure 6.3: Cost of wrong decisions. The striped and solid parts show wrong remote and local execution decisions, consecutively. Scenario1 is very asymmetric; for Castle and Ground-Table-Land almost all cost is due to wrong remote execution decisions and for Shuttle all cost is due to wrong local executions. This is expected due to asymmetric offloading rules. NWSLite beats other predictors in both scenarios.

		Wrong Local Execution Decisions			Wrong Remote Execution Decisions		
		RPF	LSQ	NWSLite	RPF	LSQ	NWSLite
Scenario1	Castle	18.76	12.90	18.15	235.28	332.29	290.32
	Shuttle	50.84	45.03	39.58	0	0	0
	Ground-Table	13.15	9.47	6.18	110.76	164.45	198.96
Scenario2	Cessna	217.47	43.66	280.86	66.77	89.85	131.85
	Venus	139.04	124.35	52.56	1464.00	1695.26	1315.97
	Ground-Table	55.72	24.23	30.66	119.87	1278.40	227.11

Table 6.3: Expected penalty for a wrong decision. The cost of a wrong decision is proportional to the complexity and size of a scene. For example, for Venus, the cost is extremely high, however for Shuttle it is very low. The expected cost is almost same for NWSLite and other predictors.

scene of 15 KBytes), the predictors always estimated that local execution was adequate and never chose remote execution.

Table 6.3 shows the expected penalty for a wrong decision; in other words, it shows how costly a wrong decision is. We compute it by dividing the total cost of wrong decisions to the number of wrong decisions n_w; that is $\sum c_i/n_w$. The results are in milliseconds for Scenario1 and in millijoules for Scenario2.

The expected penalty is not significantly different across predictors. In Scenario1, RPF had slightly lower penalty per miss, however, its effect was offset by the high number of wrong decisions. (i.e. Figure 6.2). In general, predictors have fairly close results, however, Ground-Table-Land in Scenario2, is marginal. As was explained before, this is due to the LSQ, which consistently underestimates the cost of local execution.

Table 6.3 also emphasizes the asymmetry in cost structure: In Scenario1, a wrong remote execution decision was much more expensive than a wrong local execution decision (i.e. 200 ms vs. 20 ms latency). Therefore, in settings where power consumption is not a concern, it may be beneficial to continue local execution in parallel. The same cost structure also exists in Scenario2. Here, the wrong remote execution decision penalty was 410 mJ, in contrary to the wrong local execution decision penalty which was only 107 mJ.

6.4 Summary

In this chapter, we show how NWSLite demand and supply prediction allows computation offloading to effectively optimize energy as a resource. NWSLite provides accurate predictions that indicate how much a task needs resources, and how much is available, and the "Decision Manager" decides whether to remotely or locally execute the task. NWSLite monitors the network bandwidth and latency and CPU demand and availability using an adaptive, dynamic, mixture-of-experts mechanism. After each prediction, it reconsiders the current predictor –if a predictor has less error rate than the current one, it replaces the current predictor with the one that has less prediction error. Consequently, the predictions that NWSLite produces are of higher quality than its static competitors which always use the same predictor.

The higher quality of NWSLite predictions have a clear impact on computation offloading decisions. In each scenario that we evaluate, use of NWSLite has reduced both the number of "wrong" (i.e. not beneficial) computation offloading decisions and the energy consumption.

Chapter 7

Improving Dynamic Voltage Scaling

The previous chapter demonstrated the potential of computation offloading. It showed that better predictions of resource demand and supply can improve energy efficiency (and performance) significantly, when there exists a computational server that we can offload the computation to. However, such a computational server is not always available. Furthermore, even when there is a remote computational server, it may be possible to reduce energy consumption further by scaling down performance level of local resources. Dynamic voltage scaling (DVS) is a method that lets us do so when increased task completion time caused by lower performance level is not a concern.

To minimize the effect of voltage scaling on system responsiveness, DVS policies must estimate future workload and choose the most appropriate CPU level. Accurately predicting future workload is challenging yet vital for maintaining acceptable performance. Mis-prediction can result in setting the CPU level too high, curtailing power savings, or in setting the CPU level too low, producing an unresponsive system.

Techniques that employ dynamic voltage scaling must employ efficient and accurate prediction techniques to determine how much to scale the CPU level and when. If the estimates are incorrect or the application of the optimization introduces significant overhead, the techniques may be unable to extend battery life or actually shorten it.

Prior research has concentrated on prediction techniques that estimate future CPU load as a function of previous load history [77, 3, 26, 72, 28]. More recent techniques proposed classifying tasks into different groups, each with a customized policy [17, 49]. These techniques focus on interactive tasks, such as games, word editors, notes, browsers, etc. that form a significant portion of mobile device applications. These tasks are very sensitive to lower performance, as a user perceivable slowness may be distracting and annoying for their user.

In this section, we explore how to better predict user interactivity for reducing energy consumption of mobile devices. Using the predictor that we developed in previous chapters, we design and implement an automatic voltage scaling system which we call *AutoDVS*. In Section 7.1, we discuss the user think time as a novel mechanism that can guide dynamic clock scaling of CPUs. In Section 7.2, we present our design and implementation to detect and predict future user think times. Here, we also show how to integrate our system to prior techniques that schedule CPU speed for non-interactive tasks. Section 7.3 describes how we capture user interactivity. Section 7.4 describes

evaluation metrics. Finally, Section 7.5 discusses results, and Section 7.6 gives a summary of the chapter and concludes.

7.1 Predicting User Interactivity For DVS

As we have explained in Section 2.2.2, most modern dynamic voltage scaling approaches classify interactive tasks as a separate task group and develop custom scaling policies for them. This allows keeping interactive task execution latencies at levels that are acceptable to the user. Many researchers agree that user interactive events should be completed within 50 to 100 milliseconds for providing a comfortable human-computer interaction [69].

Prior research has suggested monitoring and scheduling each interactive task automatically and individually by tracking them at the kernel level [17, 16, 49]. Interactive tasks are triggered by a user action, such as a mouse movement, or a keystroke. The aforementioned works schedule the CPU in a way that these tasks complete before the 50 milliseconds delay, whenever possible. As we have discussed before (Section 2.2.2), while this approach provides significant energy savings for interactive applications, it has the drawback of having the operating system guess when a task completes.

Guessing when a user interactive task completes is rather fuzzy. A user is probably not expecting the same interactive behavior from each user event (keystroke, mouse,

etc.,) even when they are of the same type. The intention of an interactive event may be a factor of numerous things, including:

- editing in a word editor (a keyboard event)
- starting a command using command line interface (a keyboard event)
- starting a search in a web browser (a keyboard/mouse/touchscreen event)
- action in a game (a mouse/touchscreen event)
- drawing in a back office program (a mouse/touchscreen event)
- dragging a window in window manager (a mouse/touchscreen event)
- redrawing/creating a window in window manager (a window manager triggered event)

To reschedule the CPU optimally for each event, we would have to identify the intention behind all the user events. Rather than struggle for an extremely complex model, we choose to predict interactivity as a function of inter arrival time between the user events that are generated in a particular time. In our model, we define user events in a broader term and include window redraw, create, move, focus and similar events among the ones that we monitor. Even though these events are not triggered by user directly, they are important for interactivity. The full list of events that we monitor are

Connected	A client connected to the window manager
Mouse	A mouse button / touchscreen pressed or released
Keystroke	A key pressed or released
Focus	A window received/lost focus
Region modified	A region has changed
Creation	The server has created an ID, typically for a window
Property notify	A property has changed
Property reply	The server is responding to a property value
Selection clear	A selection is cleared
Selection request	The server has queried for a selection
Selection notify	A new selection has been created
Max. window rect.	The server has changed maximum window size
COP	A communication message appeared (between GUI clients)
Window operation	A window operation (resizing, etc.)
IM Event	An input method has been used to enter non-latin text
NEvent	Number of events changed
Embed	An event used internally to implement embedded windows

Table 7.1: Interactive events that we monitor

given in Table 7.1. The list includes all the events that our windowing manager [57] allows us to capture.

Figure 7.1 shows the graphical user interface (aka GUI) event traces of a Solitaire game. We capture these while a real user was playing the game on a handheld device. The x axis shows the event identifier, which is a monotonically increasing integer starting from 0. The y-axis shows the time (in milliseconds) between two consecutive events. Solitaire receives long bursts of user input events bounded by large think times. The event burst is generated by the touch screen, during the frequent drag-and-drop operations involved in this game. The long bursts of user input events indicate poten-

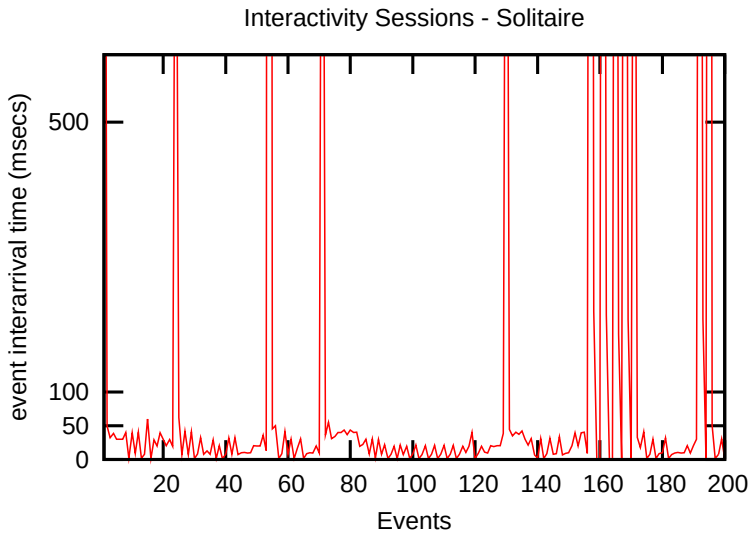


Figure 7.1: GUI event inter-arrival times for Solitaire. The y-axis shows the time (in milliseconds) between two consecutive events.

tially high CPU activity, because the events are received in quite short succession. The large think times (the spikes) indicate low CPU demand. There is a strong pattern in the graph –the typical gap between two events is 40 to 50 milliseconds, until when a large gap starts. Solitaire however, is not the only game that shows such a pattern.

Figure 7.2 and 7.3 show the event traces for a Tetrix game and a Opieplayer, which is a typical MP3 player. Tetrix receives the most user events through the keypad; this results in very short bursts of events. However, due to the nature of game, the event bursts are separated by smaller think times. If we assume that any period lasting longer than 0.5 seconds is an indication of user thinking, the median think time for Solitaire and Tetrix is 2.2 and 1.0 seconds, respectively. Opieplayer is much less interactive than

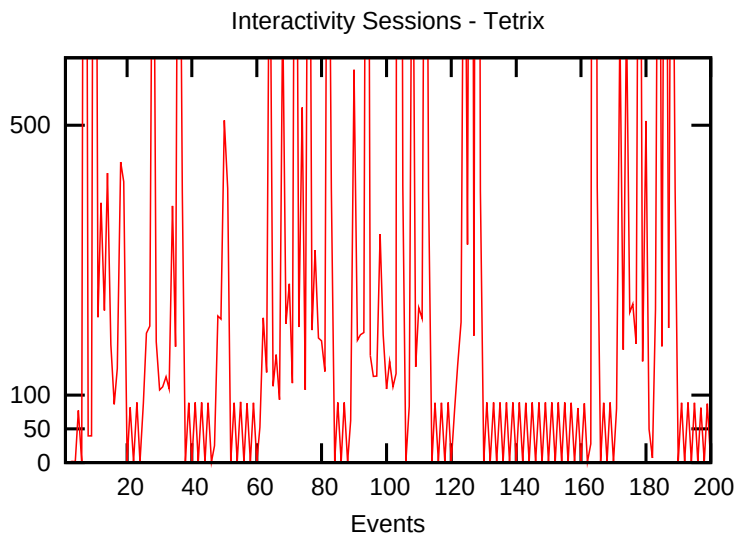


Figure 7.2: GUI event inter-arrival times for Tetrix. The y-axis shows the time (in milliseconds) between two consecutive events.

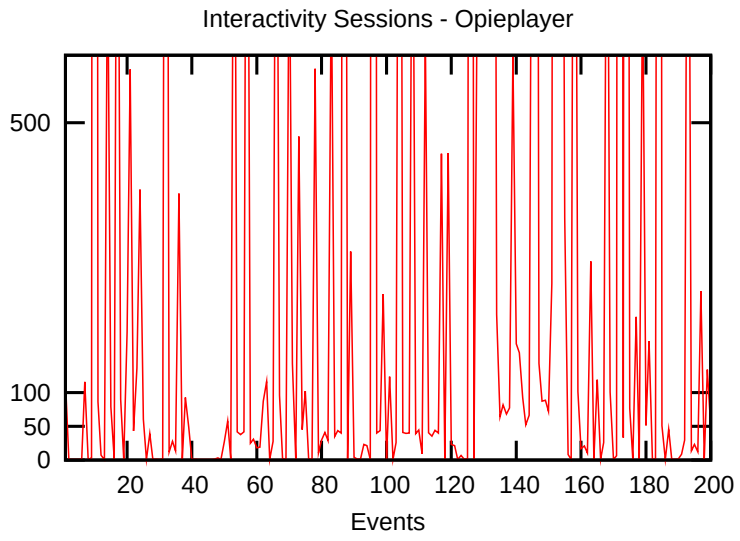


Figure 7.3: GUI event inter-arrival times for Opieplayer. The y-axis shows the time (in milliseconds) between two consecutive events.

both of those games with almost no burst of user interface events. Again, there is a pattern in the graph, however, this time, the think times are smaller, and event bursts are shorter.

In Autodvs, we monitor these GUI events to identify *interactive sessions*, the periods of high user event generation periods, in arbitrary programs. Our system employs no notion of tasks, but instead automatically infers task-like behavior, i.e., periods of time, in which the user is interacting with the device. We refer to non-interactive sessions as *think times*. In addition, we do not distinguish event types (as is done in [49]) i.e, we consider only interactive sessions regardless of which events occur within them. The goal of AutoDVS is to predict the length of user interactivity and and utilization level, which is crucial to prevent obtrusive effects of frequent clock scaling requests.

7.2 Design and Implementation

Our goal with AutoDVS is a light-weight, practical DVS system for low-end, mobile computers and their applications – without participation from the user, information from the source programs, or a priori knowledge of task length or behavior. AutoDVS transparently changes the clock frequency according to the workload activity that it senses dynamically. The key to the efficacy of AutoDVS is its categorization of application workload into two session types: interactive sessions and batch sessions. AutoDVS

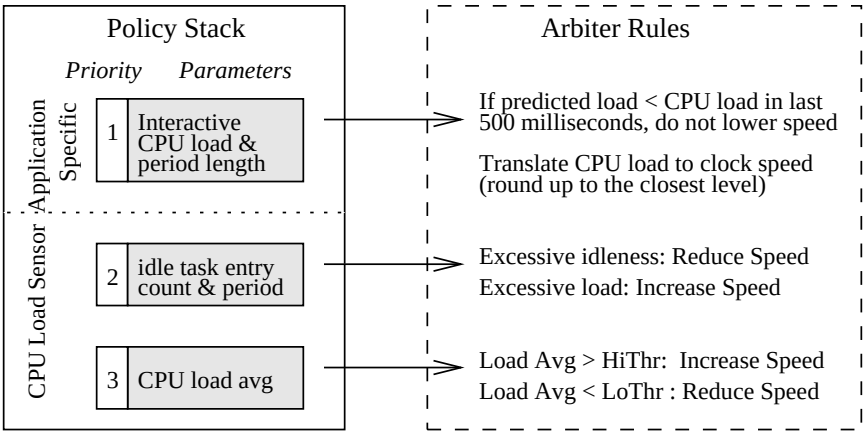


Figure 7.4: AutoDVS policy stack and arbiter rules. AutoDVS responds to policy requests according to priority (1 is highest).

then intelligently applies different, independent, and multiple scheduling policies to each session type.

For interactive sessions, AutoDVS employs a user-space policy that considers GUI events for each application. The policy predicts the duration and CPU load of each interactive session. By considering each application individually, our predictions have a better chance of capturing regular, repeating patterns within each.

For batch sessions, AutoDVS implements two different kernel-level, interval-based policies (as Linux kernel modules). The batch policies identify changes in CPU load and estimate when a CPU clock change is warranted. These CPU load predictors take a global view of the system to identify additional DVS opportunities not made apparent by the fine-grain interactive scheduler.

All CPU performance change requests issued by any policy are handled by an arbiter using the pre-defined rules that we show in Figure 7.4. The policies either request a speed change or inform the arbiter about expected CPU load and load duration. The arbiter executes requests using a priority scheme; the requests from interactive applications have the highest priority, and the requests from policies that monitor largest time-span have the lowest priority. In the case of concurrent requests, the arbiter always chooses the highest priority. However, there is one exception to this rule: If the policy that monitors excessive load detects a sudden increase in CPU demand, this request is honored first.

This division of labor across the system is key to the efficacy of AutoDVS. Together the policies are able to consider a wide range of application behaviors without much implementation complexity. By processing the requests centrally, we are able to make accurate and effective, system-wide, CPU scaling decisions that reduce power consumption without negatively impacting performance.

Moreover, this design accounts for the actual CPU change latency of the underlying device. Extant approaches to DVS scheduling assume a very low latency and allow a large number of frequency changes. For our device however, as we suggested previously, this latency can be much larger due to the overhead of maintaining other devices in the system that are synchronized with the CPU clock. We measured the switch overhead for the HP IPAQ H3800 running Linux 2.4 to be 40 milliseconds on average.

Figure 7.5 shows the timeline of a clock speed change request on this platform. We measured this period using timestamp counters. Initially, the kernel maintains a dynamic list of device drivers that must be alerted when the clock speed changes. A frequency switch request requires three traversals of this list; first to inquire if new speed is in acceptable range, and then to let device drivers initialize their hardware appropriately before and after clock change, i.e. PRECLK and POSTCLK phases. During the PRECLK and POSTCLK phases, the device drivers impose initialization delays due to hardware requirements. Even though these delays do not block the CPU, they increase the latency between clock speed request and actual change. The more devices that employ the CPU clock for timings, the longer this latency. AutoDVS does not consider sessions shorter than 50 milliseconds to account for this latency. Moreover, we can change this threshold dynamically to adapt to changing peripheral configurations.

7.2.1 Monitoring GUI Events

For each application, AutoDVS monitors user input and display updates. The former events are the direct result of user input through the keypad and touch-screen. The latter events include GUI messages such as window update and focus operations. Monitoring the display updates is important to correctly identify interactivity, e.g., when user waiting for an application to redraw the screen.

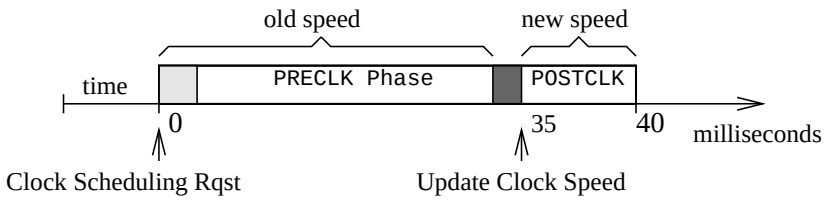


Figure 7.5: iPAQ H3800 clock scaling request timing. After receiving the request, the kernel initializes data structures and waits for initialization of hardware devices (PRECLK and POSTCLK phases). The actual clock scheduling takes under a millisecond (heavily shaded area). The clock scheduling latency on iPAQ Linux Kernel v2.4 is approximately 40 milliseconds.

In our software platform, all GUI applications are linked against a shared GUI library. We extended the `event_handler` function of this library to contain AutoDVS policies. The `event_handler` is a null (unimplemented) wrapper that receives all GUI events before any other function. Our modified `event_filter` identifies interactive sessions and interfaces to the prediction library to forecast interactive session lengths and load (we detail this process in the next subsection). We implemented a Linux new system call to provide a communication path to AutoDVS policies in the kernel.

DVS For Interactive Sessions

An interactive session starts with the arrival of an event and ends if no event is received for a period of t_p. Identifying the interactive sessions correctly is important, since presumably, the user is most sensitive to any performance loss during these peri-

ods. The value of t_p impacts the system in two ways. If t_p is too small, the algorithm might end an interactive session prematurely while the application is still processing a GUI event. If t_p is too large, AutoDVS will maintain a high CPU speed and miss opportunities for reducing energy consumption. We set t_p to be 1 second empirically: our evaluation of more than $110,000$ events on iPAQ workloads indicates that the inter-arrival time between two GUI events is less than 1 second more than 99.0% of time and that when inter-arrival time is larger than 1 second, the mean time to receive next event is 8 seconds.

When an interactive session starts, AutoDVS computes two parameters: the length of the previous session and the interactive CPU load. The computation of length (t_p) is straightforward. If e_i is the arrival time of an event such that the period between e_i and preceding event is larger than t_p. Then the length of period i is equal to $(e_{i+1} - e_i)$. We set the CPU load to be the CPU time divided by length of the period. To predict the new session length and CPU load, we employ our NWSLite predictor.

7.2.2 CPU Load Sensor

AutoDVS must also account for periods of time during workload execution that are not interactive. Most programs, even those that are primarily interactive, execute think (non-interactive or computationally intensive) periods. The CPU load sensor is responsible for these sessions. This sensor takes a global view of the system and workload,

i.e., it does not consider task-level and application-specific details. CPU load sensor employs two interval-schedulers: *CPU Load Monitor* and the *Idle Process Monitor*.

The CPU load monitor considers very large intervals (10 seconds) and averages the measured CPU load across intervals. By averaging, the monitor eliminates noise in the data and distributes *slack time* more efficiently. Slack time consists of the idle cycles during an interval when CPU utilization is less than 1. The monitor predicts that the CPU load for the next interval will be the same as it is for the current interval (this is the PAST policy used in [77, 28]). We do not use NWSLite for predicting future load since it requires floating point operations which cannot be handled in the Linux kernel. By coupling interactive task scheduling with the CPU load monitor, AutoDVS can handle both fine and coarse grain workload activities. However, we require one additional monitor to identify fine-grain behavior in non-interactive (e.g. batch and background) sessions, called the idle process monitor.

The idle process is a process that the OS scheduler executes whenever no other process in the system is runnable. The idle process monitor evaluates (then resets) idle process statistics every 500 milliseconds. The monitor considers the number of times the idle process was scheduled by the OS and its execution duration during the previous interval. We modified the Linux scheduler (sched.c) to collect and export this information as a kernel symbol. All of our modifications are light-weight, simple, and efficient and do not perceivably impact the behavior of the system.

Both monitors make CPU scaling requests to the arbitrator. The CPU load monitor uses an extension to Pering's hysteresis pair [60] to decide when to request a speed change. These values, (50,70) in prior work, act as a boundary for average CPU load. A level less than 50% indicates CPU can be scaled down and a level higher than 70% indicates CPU can be scaled up. We found empirically that the pair (60,80) works best in our actual implementation for iPAQ software. The idle process monitor requests a step increase from the arbitrator when it detects a period (500 milliseconds) in which the idle process is never scheduled. If the monitor detects that the idle process executes for over half the interval time, it requests a step decrease.

Both monitors request only single step CPU clock changes.We investigated methods such as estimating CPU cycles based on workload demand and directly switching to the most appropriate clock level (i.e. similar to [17]), however, these approaches resulted in instability, i.e. the system switched back and forth rapidly between neighboring frequency levels. The reason for this thrashing is a combination of measuring the CPU during periods of fluctuation (which impacts the measurement process) and hardware design [28].

We have empirically evaluated the efficacy of our approach by running a large number of very different workloads on an iPAQ with AutoDVS and comparative techniques. In the subsections that follow, we describe our experimental setup and the benchmark

workloads. We then define the metrics that we use in our empirical evaluation and present our results.

7.2.3 Platform Specific Design Constraints

Our device infrastructure includes five Compaq H3800 hand-held computers running Familiar Linux version 0.7.2 [15]. The H3800 is a very typical hand-held computer, with a 206MHz StrongArm CPU, and 64 Mbytes of main memory. It is capable of dynamic frequency scaling, however, it *does not yet support voltage scaling*.

To estimate power savings due to voltage scaling, we use a technique defined in prior work [17] for a similar study. We assume that the StrongArm and XScale [33] processor exhibit similar power characteristics and use published data for the XScale XSA (the system most similar to the StrongArm in terms of maximum voltage and supported frequency range), in the estimation. We approximate the voltage levels of the XScale CPU using the available frequency levels and a second degree polynomial parameterized by the XSA data:

$$v = -4 \times 10^{-7} f^2 + 0.0015 f + 0.5324 \tag{7.1}$$

To compute the corresponding StrongArm voltage levels, we use Equation 7.2 as a mapping function. That is, we linearly scale the StrongArm frequency range to the

Level	Freq. (MHz)	Estimated Voltage (mV)
1	59.0	748
2	73.7	832
3	88.5	914
4	103.2	992
5	118.0	1067
6	132.7	1139
7	147.5	1209
8	162.2	1274
9	176.9	1337
10	191.7	1397
11	206.4	1453

Table 7.2: SA1100 Parameters. We estimate the voltage levels using equations 7.1 and 7.2, and assuming that the SA1100 has the same characteristics as the XScale processor.

XScale frequency range:

$$f' = \frac{773 - 150.0}{206.4 - 59.0} \times (f - 59.0) + 150.0 \tag{7.2}$$

We present the estimated voltage levels in Table 7.2.

The StrongArm architecture requires that all of the primary peripherals be synchronous to the CPU clock [31]. This implies that all CPU scaling will impact the performance of memory, the I/O controller, DMA, the LCD controller, etc. The dependency between the CPU clock and external devices can cause significant differences between theoretical expectations (and simulated results) and practical results. For example, Grunwald et al. found that the CPU utilization changes non-linearly with respect to clock frequency, possibly due to variations in memory access cycles [28]. Another

obstacle was the LCD driver. In our evaluations, the display started vibrating making it unreadable for any speed lower than 103MHz. Thus we had to eliminate three lowest frequencies.

The window manager that we run on the devices is Opie [57] version 1.0.2. Opie is an open-source graphical user interface designed for Sharp Zaurus and Compaq hand-held computers. It is a full-fledged GUI comparable to commercial versions in both appearance and features. The available Opie applications include Calendar, Contacts, Drawpad, a multimedia player, a wide range of games, etc.

7.3 Collecting User Interactivity Traces

We evaluate AutoDVS below using two different scenarios. (1) Interactive: Running GUI applications; and (2) Concurrent: Interactive and soft-real time applications running together.

To evaluate and compare the performance of interactive applications, we collect a set of usage traces and extracted event and timestamp information. We then monitor the performance of AutoDVS while replaying the events in real time. Thus, *our results also include the overhead of clock switching and all AutoDVS functionality*.

To collect the usage traces, we have installed Opie on several Compaq H3800 hand-held computers and have distributed them to graduate students in our department. We

alert the students that we are capturing all events and ask them to use the hand-helds as their own as normally as possible and to reboot them periodically (to end the session).

We modify the iPAQ software to enable trace collection in three ways: We (1) disable network connectivity; (2) modify random number generators to use a fixed seed; and (3) program each to clear all user state information after every reboot. These changes were necessary to eliminate as much non-determinism as possible so that we could re-generate the user events in the correct order during experimentation.

To capture the events, we instrument the Linux kernel at the I/O driver level. Our system captures all events generated by the touch-screen, the keypad, and the joy-pad using a microsecond timestamp. We save the identification information for captured events in RAM and copies them to permanent storage immediately prior to shut-down, to prevent any excessive overhead. The time and space overhead for event trace collection is small. Each event requires a total of 20 bytes: 8 bytes for the timestamp and 12 bytes for event type and attributes. Since we capture events at the I/O device driver, we can read the current time directly from Linux kernel data structures and no system calls are required.

To replay the captured events, we have developed a Linux kernel module. The module initiates events from a list in memory using a microsecond resolution timer.

The events describe user behavior from boot-up to shutdown. Some of the event traces that we captured are not useful; they are either too short, broken, i.e., depen-

Trace	Event Count (ETime@206MHz)	Description
DrawPad-1	23100 (915.4s)	Drawing random pictures
General-1	3688 (448.1s)	General use including calendar, contacts and games
Solitaire-1	8700 (756.4s)	Multiple Solitaire games
Tetrix-1	6936 (583.8s)	Tetrix
Tetrix-2	1342 (210.1s)	Tetrix - very short and slow
Checkers-1	1238 (205.1s)	Checkers - medium difficulty
Checkers-2	1214 (265.7s)	Checkers - maximum difficulty
Checkers-3	2490 (1076.4s)	Checkers - maximum difficulty

Table 7.3: AutoDVS evaluation benchmarks and event traces. We gather the traces using instrumented versions of the system while different users exercised the iPAQs. The name of each trace reflects the application that was dominant during the usage period.

dent on user created files, or too similar. Overall, we employ the traces described in Table 7.3. The second column is the number of events in the trace and the total time (seconds) for the real time play-back at maximum performance (206MHz). We refer to each event trace using the name of the application that was active most often. The first four traces describe more general-use applications and include multiple program types. The last four traces are exclusively games.

To evaluate soft real-time applications we use Madplay, an open-source, high quality, MP3 decoder [50]. We interface Madplay to the GNOME Enlightened Sound Daemon (ESD), to enable on-line playback. The input file encoding rate is 56Kbits/sec.

7.4 Evaluation Metrics

We have evaluated the impact of AutoDVS using three different metrics: *energy factor*, *stall rate*, and *stall magnitude*. Energy factor measures energy consumption with respect to execution at full CPU performance. The stall rate and magnitude metrics describe the degradation in interactive performance due to CPU scaling.

We compute energy consumption using the `energy factor` (EF) as defined in [17]. EF is the ratio of energy used by the scaled workload to energy used when workload is processed at full speed. That is,

$$EF = \frac{\sum_{i=1}^{n} v_i^2 f_i t_i}{v_{MAX}^2 f_{MAX} T} \tag{7.3}$$

where v_i and f_i are the voltage and frequency of each period of time (t_i) between two frequency scaling operations and T is the execution time of benchmark at full CPU performance level. We use the frequency and voltage levels that are given in Table 7.2. EF is unit-less and measures the energy consumption of the CPU only.

To compute performance loss, we record the execution time of each interactive event. Specifically, we assume that execution of an event starts when it arrives at the window manager. We define event completion time using the approach described in [49]: The execution of an event ends when the idle task is entered and no I/O is ongoing. Even though this method can be imprecise (i.e. it might occasionally mis-classify events as completed), other (extant) approaches (described in Section 2.2.2) are highly

complex and can adversely affect the performance of monitored system. We discuss the impact of using this method when we present our results.

We assume that an event misses its deadline if its execution time is larger than user perception threshold, i.e. 50 milliseconds [17]. Given that d is the deadline, and m_k is the execution time of an event which missed its deadline, the stall rate (SR) is:

$$SR = \frac{\sum_{k=1}^{n}(m_k - d)}{T} \tag{7.4}$$

SR is a unit-less metric that measures the user perceived performance loss during the execution of a benchmark. However this metric does not indicate how much the user must wait for a stalled system. To measure this, we use stall magnitude (SM):

$$SM = \frac{\sum_{k=1}^{n}(m_k - d)}{n} \tag{7.5}$$

SM measures the average stall time due to interactive events that miss their deadline. The unit of SM is seconds.

Finally, to measure the quality of music playback when we consider concurrent workloads, we count the number of buffer underruns that occur in the ESD sound driver. Buffer underruns indicate that the MP3 decoder has missed its deadline for replenishing consumed data. When this happens, the sound driver fills the gap by repeating the most recent data. Each buffer underrun is perceivable by user, however the degree to which it degrades the overall quality of the experience, is a matter of personal taste. We

therefore, treat each buffer underrun as equally undesirable and disregard the duration of each individual underrun period.

7.5 Results

We compare AutoDVS to two other policies: MAX, in which the CPU is set to the highest level (for maximum performance), and IDEAL, in which we employ an ideal (oracle-based) CPU speed. IDEAL is not a realistic policy; it always chooses a clock speed such that its performance degradation is at the level of AutoDVS or less. To accomplish this, IDEAL uses future information: If IDEAL is worse than AUTODVS in terms of both performance metrics (i.e: SR and SM), or if the quality of sound playback is inadequate (i.e. more than 10 buffer underruns), then IDEAL switches to a higher clock frequency. We limited IDEAL choices to 132MHz, 176MHz and 206MHz to limit the search space.

We have experimented with two different scenarios. (1) Interactive: Running GUI applications; and (2) Concurrent: Interactive and soft-real time applications running together. We describe the results from each of these scenarios in the following subsections.

7.5.1 Interactive Workloads

We first evaluate AutoDVS in terms of performance degradation during the execution of interactive applications. The IDEAL policy has an advantage in this dataset; our empirical evaluations show that most interactive tasks require only a fraction of the maximum CPU power. A flat policy of 132MHz will provide adequate performance. The question we want to answer is this: Can AutoDVS achieve similar energy savings and still maintain a high level of responsiveness?

Figure 7.6 compares AutoDVS and IDEAL in terms of energy factor, stall magnitude, and stall rate, from left to right. For all sub-figures, a lower bar indicates a better performance. For the first sub-figure, a lower bar indicates reduced energy consumption, for the last two sub-figures, a lower bar indicate a better response time. We label the bars with the first three letters of the event trace, these are **Dra**wing, **Gen**eral, **Sol**itaire, **Tet**rix**-1**, **Tet**rix**-2**, **Che**ckers**-1**, **Che**ckers**-2** and **Che**ckers**-3**, from left to right. For example, for the Drawing benchmark, the AutoDVS policy saved almost 60% of energy with a 10% stall rate and approximately 240 milliseconds stall magnitude (i.e. mean stall time due to interactive events that miss their deadline).

AutoDVS enables significant performance benefits for the first six benchmarks. While keeping the stall rate under 10% of total execution time, AutoDVS reduces energy consumption 30–66% (49% on average). In general, energy consumption is proportional to the performance requirements of benchmarks. For example, for Gen-1 and

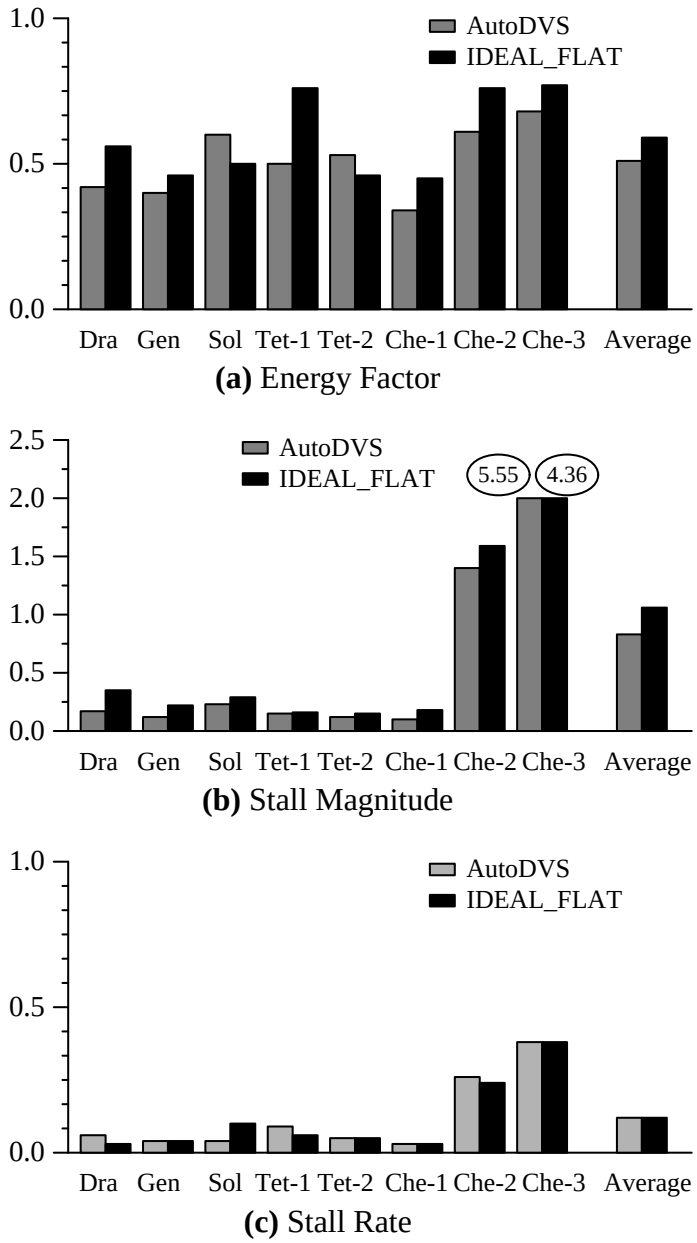


Figure 7.6: Performance of AutoDVS and IDEAL for interactive workloads. A lower bar indicates better performance. AutoDVS reduces energy consumption by 10% over IDEAL (Figure (a)), Overall energy savings due to AutoDVS is 49% on average. AutoDVS average stall time (Figure (b)) is lower than IDEAL even though the stall rates are the same (Figure (c)). The circled values in (b) are the actual data values – that are cut off in the graph for clarity.

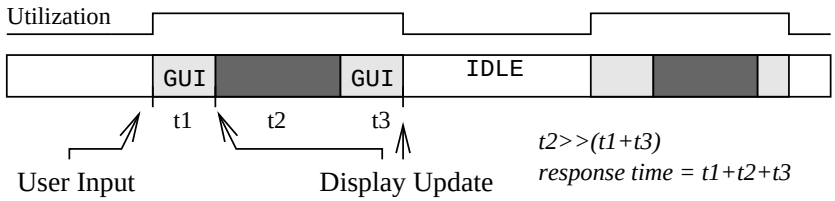


Figure 7.7: CPU utilization in Checkers.In Checkers, CPU utilization is highly periodic and changes in a boolean fashion. Each user input triggers two GUI update events (light gray boxes), and a computationally intensive task (dark gray box). This periodic behavior reduces voltage scaling opportunities.

Che-1, which both include game sessions at novice levels, the savings are the greatest. For the first six benchmarks, IDEAL uses 176MHz for Tet-1 only and 132MHz for the rest. In general, IDEAL uses almost 10% more energy to maintain the stall rate of AutoDVS. However, AutoDVS is able to predict CPU demand accurately to reduce stall time. On average, AutoDVS achieves a 35% improvement over IDEAL.

Che-2 and Che-3 exhibit different behavior patterns than the other benchmarks. Both of these traces are game sessions at the highest difficulty level. As Figure 7.7 shows, their workload is very regular and highly computationally intensive. Each user input triggers a computationally intensive task which is followed by a long idle period (think time). The NWSLite is unable to predict this behavior accurately. It is possible to estimate such behavior using techniques such as spectral analysis [9], however, we avoid such algorithms in NWSLite due to their high computational (floating point)

cost. Despite some prediction error, AutoDVS enables energy consumption for Che-2 and Che-3 that is 36% lower than MAX and 15% lower than IDEAL.

7.5.2 Concurrent Workloads

We next investigate how AutoDVS performs when multiple applications are running concurrently on an iPAQ. In particular, we replay the event trace while running the Madplay MP3 decoder (music player) in the background. For each event trace, we start collecting the measurement statistics when the two tasks begin executing events concurrently; we continue measuring until Madplay terminates. There are three short-traces that end earlier than Madplay, Tet-2, Che-1 and Che-2. For these traces, Madplay is the single task for 45%, 42% and 33% of total evaluation time, respectively. The Madplay playback length is 424 seconds.

The opportunities for CPU scaling are reduced when we execute multiple programs concurrently. The question that we are interested in is whether it is possible to extract any energy savings without hurting performance.

Figure 7.5.2 compares the performance of AutoDVS to IDEAL using the same methodology as the previous subsection. AutoDVS is able to save 31% of the energy consumption over MAX on average. The energy savings of IDEAL is 20%. IDEAL chooses 176MHz for all but Che-2 and Che-3. For these two traces, IDEAL uses the

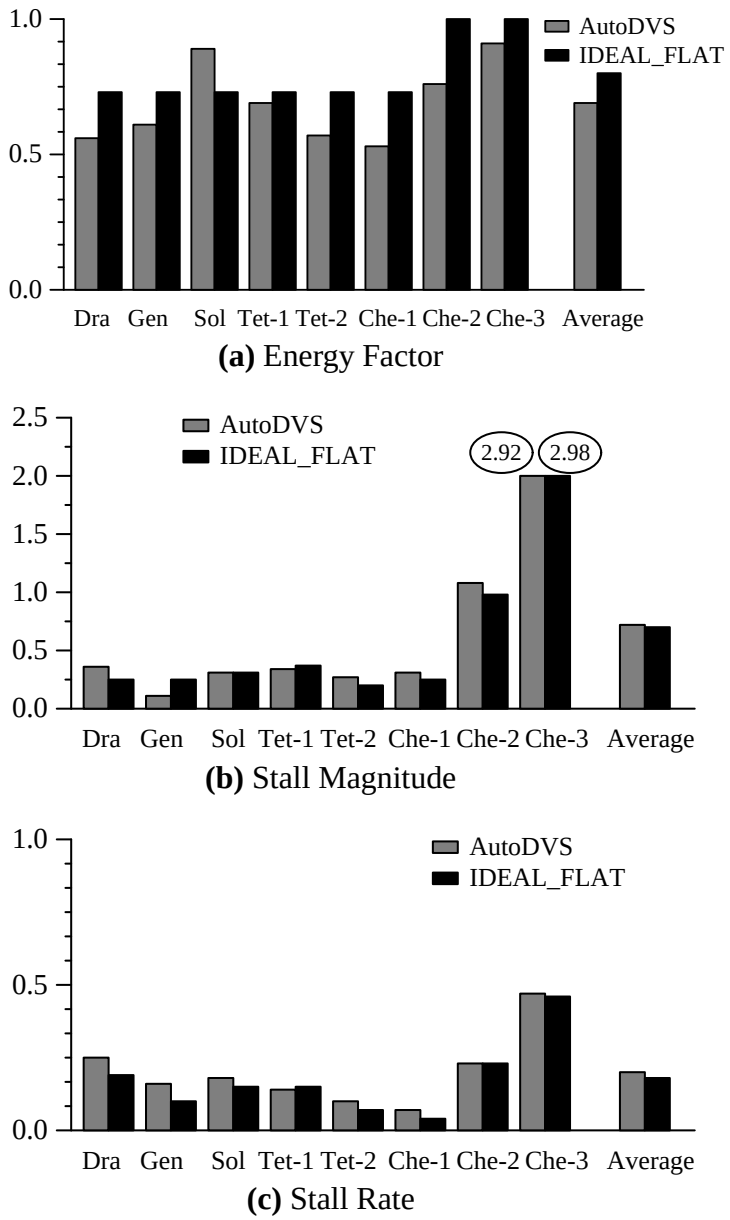


Figure 7.8: Performance of AutoDVS for interactive and soft-real time workloads, running concurrently. A lower bar indicates better performance. The overall energy savings is 31% for AutoDVS and 20% for IDEAL (Figure (a)). Figures (b) and (c) report stall magnitude and rate, respectively. The circled values in (b) are the actual data values – that are cut off in the graph for clarity.

maximum level (206MHz). The savings are small for traces with high computational requirements, e.g., Che-3 and Sol.

At the 132MHz (lowest) level, the buffer underruns for Madplay are very high (122–2781) for all benchmarks. At 176MHz, there are fewer than 3 underruns. However, IDEAL must switch to 206MHz (the highest level) for Che-2 and Che-3 to achieve the same performance as AutoDVS. For example, Che-2 at 176MHz imposes a 8% larger stall rate and an average stall time that is 179 milliseconds worse than AutoDVS. The performance loss margin is greater for Che-3. The buffer underrun count is always less than 3 for AutoDVS. For MAX, buffer underruns are always 0.

In the concurrent workload results, the only anomalous case in which IDEAL outperforms AutoDVS in terms of energy consumption is for Solitaire. AutoDVS uses 13% more energy than its competitor to achieve approximately same performance level. Solitaire is unique in that most of the GUI events are mouse drag/drop events; the other benchmarks use keypad, joypad, or touchscreen keyboard for most of the data input. Each drag and drop generates a sudden burst of GUI events (i.e. window update messages and mouse movements). Consequently, this sudden burst is accompanied with a jump in CPU demand. NWSLite immediately chooses the most aggressive forecaster, often over-estimating the CPU load for the next interactive session. Even though the CPU load sensor policies correct the over-estimation afterwards, some portion of en-

ergy is wasted during this period. Moreover, an increase in clock speed does not reduce the stall magnitude significantly since it is already very low.

We can address this problem in two ways:(1) by classifying drag and drop events separately from other GUI events and reducing the weight of events generated in bursts, or by (2) shutting down predictors that consistently over-estimate. However, we do not evaluate any of these alternatives in the scope of this thesis.

7.5.3 Integrating PACE

As a final experiment, we investigate the efficacy of extending AutoDVS to conserve additional energy on platforms that have very low voltage switch latency. To investigate this, we have incorporated extant, efficient, implementation of the PACE algorithm [48, 49], called Practical Pace (PPACE) [84], into AutoDVS.

PACE is a technique that computes optimal energy savings when *continuous* CPU scaling is possible. PACE computes CPU speed as a function of completed work and gradually increases the CPU frequency as the task nears its deadline. PPACE extends PACE to handle discrete CPU scaling levels and uses a polynomial time approximation of PACE that is computationally efficient but does not always find the optimal solution.

We investigate the impact of integrating PPACE into AutoDVS. We employ simulation for these experiments (unlike in our previous experiments) since an actual, online implementation of PPACE is currently not feasible due to three primary reasons. First,

Parameter	Value	Description
D	50 Msecs	Task deadline
WC	6.192 Mcycles	Worst-case execution cycles
r	6	Number of transition points
f	(103.2 - 206.4) MHz	StrongArm Clock frequency
s	[WC-1] / r	Transition period -evenly spaced
ϵ	0.05	Trim error parameter

Table 7.4: PPACE simulation parameters

extent hand-held devices impose a very high switch latency. Second, the computational requirements of PPACE are high and consume significant resources in modern devices. Third, the computation of cumulative distribution function requires off-line information.

Despite these limitations, we are interested in understanding the potential of coupling PPACE and AutoDVS for interactive programs. For these experiments, *we consider the energy consumption due to GUI events alone*. Our results indicate the potential energy savings during the execution of pre-deadline cycles. By definition, the PACE algorithms do not reschedule post-deadline cycles.

To integrate PPACE into AutoDVS, we extended AutoDVS with an additional API through which it consumes off-line information, task deadlines, and the worst-case execution times (WCETs) of tasks from the program. Table 7.4 shows the parameters we use to evaluate PPACE in AutoDVS. To determine the WCET in cycles, we use the CPU demand of 99 percentile of GUI tasks which is equal to 6.192 Mcycles. We

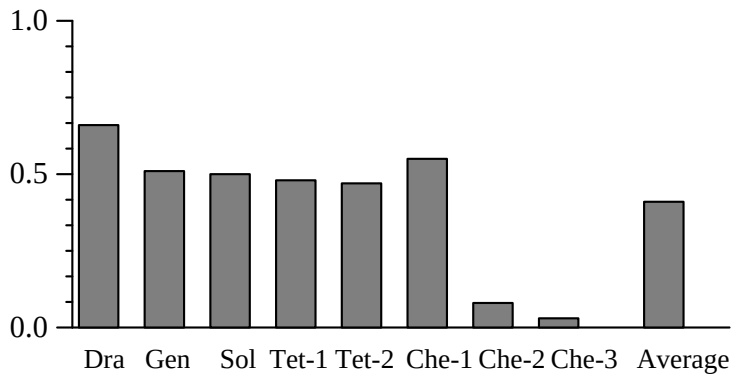


Figure 7.9: *Simulated* energy savings ratio with respect to AutoDVS. These results are different from all those prior in that we obtain them through simulation and consider only GUI events. Higher bars are better. On average, incorporating PPACE results in a potential 41% decrease in energy consumption of GUI events when the event deadlines and WCETs are known a priori.

limited the number of clock speed transitions to 6, placing them evenly in the range [1,WCET]. Even though we use a smaller number of transitions than were used in the previous PPACE, our implementation provides a higher resolution than the original implementation since we use a much smaller WCET. Xu et al. uses $WC = 500$ Mcycles with 100 transition points – this corresponds to a transition point approximately every 5 Mcycles. In contrast, we place a transition point at approximately every 1 Mcycles.

Figure 7.9 shows the energy savings ratio for GUI events when we reschedule CPU speed using PPACE – relative to AutoDVS (not MAX as in prior graphs). Higher bars are better. The data indicates that using PPACE with AutoDVS can potentially enable significant energy savings. Our results indicate that for most of our event traces, the energy consumption of GUI events can be decreased by over 50%. Che-2 and Che-3

are exceptions to general trend. For these two cases, the savings are less than 10%. We find that on average, PPACE reduces the energy consumption of GUI events by 41%.

7.6 Summary

In an effort to produce an automatic DVS system for a popular hand-held device, we developed a set of Linux extensions that couple and extend a number of extant approaches. Our system is called AutoDVS and is very flexible and extensible. Each of the DVS algorithms that we used for different workload behaviors can be replaced with others. We intend for it to be used by researchers interested in investigating, empirically evaluating, and comparing DVS algorithms on iPAQ hand-helds using popular and general-purpose hand-held software.

Our results indicate that AutoDVS can reduce power consumption significantly for a wide range of application types executed alone or concurrently. On average, AutoDVS reduces power consumption by 49% for interactive tasks, and by 31% for concurrent workloads. AutoDVS enables these results automatically and transparently for a wide range of real applications. The key to enabling these power reductions is the use of interval schedulers that capture computationally intensive and idle periods in the workload and accurate time series prediction to estimate the duration of application-specific interactive sessions. The combination of these techniques enables AutoDVS to infer ac-

curately task-level behavior from applications, workloads, and concurrent workloads, and to adapt the clock speed appropriately.

Finally, with AutoDVS, we show that event arrival frequency (interarrival times between events) is a useful metric for measuring user interactivity. We do this without requiring any modification to user applications, simply by monitoring the events that the window manager receives and broadcasts.

Chapter 8

Conclusions

In this dissertation, we explored better resource measurement and prediction support for enabling more effective power management in embedded, resource-restricted computers. We discussed the significance of prediction in dynamic voltage scaling and computation offloading efficacy. In this work, we developed dynamic, adaptive techniques that can provide such accuracy with low resource consumption.

As the power management techniques become more mature, the need for more effective prediction methods will increase significantly. At present most computation offloading and dynamic voltage scaling techniques use static, parameterized prediction techniques that are tedious to develop and parameterize. While the static techniques are highly accurate for the workload (dataset) that they are parameterized for, they may not capture the changes in workload characteristics. Here, we discussed the design of an adaptive, dynamic prediction utility for embedded systems, and we validated its efficacy using a large dataset that is collected from real systems. In addition, we discussed

the challenges of measuring energy consumption at run-time, and we proposed a run-time power measurement technique which we validated using the Stargate computer that we have.

We provide the following contributions in this dissertation.

- A run-time, fine grain task power measurement technique. The technique that we developed can predict power measurement with great accuracy (3.8% to 4.6% error rate). It updates the model at run-time using battery monitor feedback.

- A non-parametric prediction tool that make accurate forecasts of future application and resource behavior, wireless bandwidth, CPU, network bandwidth and latency. Our tool surpasses, or at least matches that of commonly used exponential smoothing and least squares predictors, without any tedious parameterization. Our tool uses only 55 floating point operations for each prediction.

- Demonstration of significance of prediction accuracy in computation offloading. Using simulation on real data, we demonstrated that better prediction leads to a significant improvement in energy savings. Our evaluations show that NWSLite can reduce wasted energy (that is due to wrong decisions) 27% to 56% in comparison to its static parameterized competitors.

- Application of better prediction to dynamic voltage scaling. User think time can be used as an important opportunity to scale CPU performance and voltage

level. We measure user think time directly inside window manager, without any change to application source code, and then we use these measurements to predict future think times using our predictors. By combining our mechanism to already existing DVS methods, we show significant energy savings (31% to 49%) are possible.

In concluding remarks, we discuss potential improvements to proposed techniques and future research directions that these techniques enable.

8.1 Directions For Future Research

The capability of accurately measuring task power consumption can have a profound effect on the design and implementation of power aware operating systems. Once the operating system knows its energy budget and how much energy is required by different applications, tasks and threads, it can allocate energy much more efficiently in order to satisfy user requirements (such as battery life, application performance). Future operating systems, such as the recently proposed ECOsystem [87], micro-manage energy consumption for optimum battery life. In ECOsystem, the authors propose allocating energy to tasks using a priority mechanism that is set by the user. As the tasks use computational resources, CPU, I/O and network interface, they pay their share of energy consumption to the operating system, in units of Joules. For the realization of

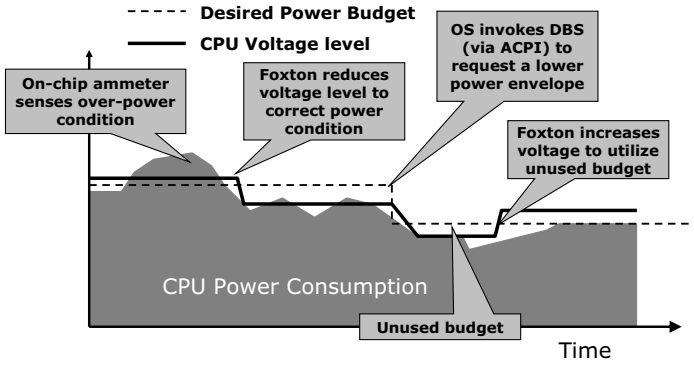


Figure 8.1: Power management using Foxton technology. The y-axis is the CPU power consumption. The A/D converters that exist on CPU measure power continuously and scale CPU voltage/frequency accordingly to keep power consumption in a certain budget. The operating system uses DBS (demand-basing switching) interface to convey the power consumption limit.

these systems, run-time power measurement capability is key (which this thesis discusses). Therefore, one of the future directions that this dissertation enables is the design and implementation of power aware operating systems that can measure (and optimize) software energy consumption. Combining our system with ECOsystem may be a good starting point for achieving this goal.

Since our work is a first step to run-time power measurement of the full system, there are a number of areas that must be improved. First of all, we need to improve hardware capability to provide more accurate power measurements. As we demon-

strated, current battery monitors are not as accurate yet. Luckily however, there are new projects which, as a side effect, can lead to technologies that facilitate such accurate battery sensors. One such project is Intel Foxton [61]. In this project, the researchers couple the existing CPUs with very high speed and very accurate A/D converters that measure power consumption of CPU at real time, and with great accuracy. The micro-controller that controls these A/D converters automatically adjust the CPU voltage level and frequency using these instantaneous power consumption measurements such that it keeps the power consumption under a certain budget. The operating system determines what this budget is. Figure 8.1 explains this concept in more detail.

There are two ways that Intel Foxton (and similar future projects) can improve run-time power measurement support. First, if Foxton microcontroller can be enhanced in a way akin to hardware performance monitors, operating system can use this interface to measure CPU power consumption in real time, for any application, task, or even a procedure. Second, such projects can enable production of highly accurate battery monitoring sensors that can measure full system energy consumption with great accuracy. By coupling these measurements with the models that we propose, it will be possible for the OS to assess the energy cost of tasks and operating system threads. In measuring full system energy consumption, a significant difficulty is I/O devices. In this thesis, we proposed using software counters for this purpose and demonstrated their effectiveness using the communication interface. However, there is still a lot to

do. Many of the I/O states (and their power behavior) are transparent to software counters, since there is a firmware/microcode layer (mostly proprietary) that controls these devices. In addition, since I/O is mostly asynchronous to application execution, its measurement and mapping to applications accurately is still an open question. We believe that further investigation of I/O power consumption is highly justified for the design and implementation of future operating systems.

Bibliography

[1] A. Balachandran, G. Voelker, P. Bahl, and P. Rangan. Characterizing user behavior and network performance in a public wireless lan. In *ACM International Conference on Measurement and Modeling of Computer Systems*, 2002.

[2] R. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *International Conference on Mobile Systems, Applications, and Services*, 2003.

[3] S. A. Banachowski and S. A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Proceedings of Multimedia Computing and Networking*, January 2002.

[4] Rechargeable battery/systems for communication/electronic applications. http://www.acq.osd.mil/ott/natibo/docs/BatryRpt.pdf.

[5] B.D.Cahoon. *Effective Compile-Time Analysis For Data Prefetching In Java*. PhD thesis, University of Massachusetts at Amherst, 2002.

[6] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley and Sons, 2003.

[7] W. L. Bircher, M. Valluri, J. Law, and L. K. John. Runtime identification of microprocessor energy saving opportunities. In *International Symposium on Low Power Electronics and Design*, 2005.

[8] G. Bottomley and S. Alexander. A novel approach for stabilizing recursive least squares filters. *IEEE Transactions on Signal Processing*, August 1991.

[9] P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer-Verlag, 2002.

[10] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, UW Madison Computer Sciences, June 1997.

[11] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[12] G. Contreras and M. Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *International Symposium on Low Power Electronics and Design*, 2005.

[13] Dallas Semiconductors. *DS2760 Data Sheet*. http://pdfserv.maxim-ic.com/arpdf/DS2760.pdf.

[14] J. Eager. Advances in rechargeable batteries spark product innovation. In *Silicon Valley Computer Conference*, August 1992.

[15] Familiar Web Page – http://www.handhelds.org.

[16] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for Linux. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation OSDI*, December 2002.

[17] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. *Wireless Networking*, 8(5):507–520, 2002.

[18] J. Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, Carnegie Mellon University, Dec. 2001.

[19] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems(HotOS-VIII)*, pages 61–66, Germany, 2001.

[20] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *International Conference on Distributed Computing Systems (ICDCS '02)*, pages 217–226, 2002.

[21] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[22] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[23] R. Freund and P. Minton. *Regression Methods, A tool for Data Analysis*, volume 30. Marcel Dekker, 1979.

[24] Game of Life – http://daugerresearch.com/vault/parallellife.shtml.

[25] GLVU source code and documentation, Feb 2002. http://www.cs.unc.edu/ walk/software/glvu/.

[26] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *ACM International Conference On Mobile Computing And Networking (MoBiCom)*, pages 13–25, 1995.

[27] The grid application development software project (GrADS). "http://hipersoft.cs.rice.edu/grads/".

[28] D. Grunwald, P. Levis, C. Morrey, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. In *Operating System Design and Implementation(OSDI)*, pages 73–86, October 2000.

[29] M. R. Guthaus, J. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, dec 2001.

[30] Hewlett-Packard. *Compaq iPAQ Pocket PC H3800 Series Reference Guide*, March 2002. Document Part Number: 253194-002.

[31] Intel. *StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001. Order Number:278240-004.

[32] Intel. *Intel PXA-270 Processor: Electrical, Mechanical and Thermal Specification*, June 2004. Order Number:280002-006.

[33] Intel Corporation. *XScale*. www.intel.com/design/intelxscale/.

[34] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *WWC '03: Proceedings of the Sixth International Workshop on Workload Characterization*, 2003.

[35] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO '03: Proceedings of the 36th ACM/IEEE International Symposium on Microarchitecture*, 2003.

[36] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *HPCA '06: Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, 2006.

[37] I. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[38] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *International Symposium on Low Power Electronics and Design*, pages 135–140, 2001.

[39] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykirsnan, M. J. Irwin, and A. Sivasubramaniam. vec: virtual energy counters. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop On Program Analysis For Software Tools And Engineering (PASTE)*, 2001.

[40] M. Kim and B. Noble. Mobile network estimation. In *Mobile Computing and Networking*, pages 298–309, 2001.

[41] U. Kremer, J.Hicks, and J.M.Rehg. A compilation framework for power and energy management on mobile computers. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, Aug. 2001.

[42] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[43] C. Krintz and B. Calder. Using annotation to reduce dynamic optimization time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, June 2001.

[44] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[45] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, 1997.

[46] T. Li and L. John. Run-time modeling and estimation of operating system power consumption. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2003.

[47] X. Liu, P. Shenoy, and M. Corner. Chameloen: Application controlled power management with performance isolation. Technical Report 04-26, Department of Computer Science University of Massachusetts, 2004.

[48] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–61. ACM Press, 2001.

[49] J. Lorch and A. Smith. Using user interface event information in dynamic voltage scaling algorithms. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation Computer and Telecommunications Systems*, pages 46–55, October 2003.

[50] Madplay – http://www.underbit.com/products/mad/.

[51] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 721–725. ACM Press, 2002.

[52] R. Myers. *Classical and Modern Regression with Applications*. PWS-KENT Publishing Company, 1989.

[53] D. Narayanan. *Operating System Support for Mobile Interactive Applications*. PhD thesis, Carnegie Mellon University CMU-CS-02-168, Aug. 2002.

[54] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *International Conference on Mobile Systems, Applications, and Services*, 2003.

[55] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *sixteenth ACM symposium on Operating systems principles*, pages 276–287. ACM Press, 1997.

[56] The Network Weather Service Home page – http://nws.cs.ucsb.edu.

[57] OpenZaurus Web Page – http://www.openzaurus.org/web.

[58] PAPI – http://icl.cs.utk.edu/papi.

[59] M. Pedram and J. Rabaey. *Power Aware Design Methodologies*. Kluwer Academic Publishers, 2002.

[60] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. International Symposium on Low Power Electronics and Design*, pages 76–81, Aug. 1998.

[61] C. Poirier, R. McGowen, C. Bostak, and S. Naffziger. Power and temperature control on a 90nm itanium family processor. In *International Solid State Circuits Conference (ISSCC)*, San Francisco, CA, February 2005.

[62] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Annual International Conference On Mobile Computing And Networking*, pages 251–259. ACM Press, 2001.

[63] Q.O.Snell, A.Mikler, and J.L.Gustafson. Netpipe: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[64] A. Rudenko, P. Reiher, G.Popek, and G.Kuenning. The remote processing framework for portable computer power saving. In *ACM Symp. Appl. Comp.*, San Antonio,TX, Feb. 1999.

[65] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, Jan. 1998.

[66] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[67] L. Shang, A. Kaviani, and K. Bathala. Dynamic power consumption in Virtex-II FPGA family. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, pages 157–164, 2002.

[68] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis. Tina: a scheme for temporal coherency-aware in-network aggregation. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 69–76, New York, NY, USA, 2003. ACM Press.

[69] B. Shneiderman. *Designing The User Interface: Strategies For Effective Human Computer Interaction*. Addison-Wesley, 1998.

[70] Wireless LAN Traces from ACM SIGCOMM'01 – http://ramp.ucsd.edu/pawn/sigcomm-trace/.

[71] A. Sinha. *Energy Efficient Operating Systems and Software*. PhD thesis, Massachusetts Institute of Technology, 2001.

[72] A. Sinha and A. P. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of the The 14th International Conference on VLSI Design (VLSID '01)*, page 221. IEEE Computer Society, 2001.

[73] A. Sinha and A. P. Chandrakasan. Jouletrack: a web based tool for software energy profiling. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 220–225, New York, NY, USA, 2001. ACM Press.

[74] M. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo00)*, Jan. 2000.

[75] Sundials: Suite of nonlinear and differential algebraic equation solvers – http://www.llnl.gov/CASC/sundials/main.html.

[76] Y. Wang, E. Y. Chang, and K. P. Cheng. A video analysis framework for soft biometry security surveillance. In *VSSN '05: Proceedings of the third ACM international workshop on Video surveillance & sensor networks*, pages 71–78, New York, NY, USA, 2005. ACM Press.

[77] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[78] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02: Proceedings of the 2002 International Conference On Compilers, Architecture, And Synthesis For Embedded Systems*, pages 238–246, New York, NY, USA, 2002. ACM Press.

[79] A. J. Willmott. Radiator source code and online documentation, Oct 1999. http://www.cs.cmu.edu/ ajw/software/.

[80] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, Jan. 1998.

[81] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 1999.

[82] N.-S. Woo. Promises and challenges of mobile embedded system:: an industry perspective. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 3–3, New York, NY, USA, 2006. ACM Press.

[83] Gartner Inc, press release –http://www.gartner.com, October 2006.

[84] R. Xu, C. Xi, R. Melhem, and D. Moss. Practical pace for embedded systems. In *Proceedings of the fourth ACM international conference on Embedded software*, pages 54–63. ACM Press, 2004.

[85] P. Young. *Recursive Estimation and Time-Series Analysis*. Springer-Verlag, 1984.

[86] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[87] H. Zeng, C. Ellis, A. Leveck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[88] Z.Li, C.Wang, and R.Xu. Computation offloading to save energy on handheld devices:a partition scheme. In *Proc. of International Conference on Compilers,Architectures and Synthesis for Embedded Systems (CASES)*, pages 238–246, 2001.