Energy Efficient Operating Systems and Software

by

Amit Sinha

Bachelor of Technology in Electrical Engineering
Indian Institute of Technology, Delhi, 1998

Master of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2000

Submitted to the Department of Electrical Engineering
and Computer Science in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
August 2001
[September 2001]

© 2001 Massachusetts Institute of Technology. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August, 2001

Certified by ....
Anantha Chandrakasan
Associate Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 01 2001
LIBRARIES
BARKER

Energy Efficient Operating Systems and Software

by

Amit Sinha

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science.

Abstract

Energy efficient system design is becoming increasingly important with the proliferation of portable, battery-operated appliances such as laptops, Personal Digital Assistants (PDAs) and cellular phones. Numerous dedicated hardware approaches for energy minimization have been proposed while software energy efficiency has been relatively unexplored. Since it is the software that drives the hardware, decisions taken during software design can have a significant impact on system energy consumption. This thesis explores avenues for improving system energy efficiency from application level to the operating system level. The embedded operating system can have a significant impact on system energy by performing dynamic power management both in the active and passive states of the device. Software controlled active power management techniques using dynamic voltage and frequency scaling have been explored. Efficient workload prediction strategies have been developed that enable just-in-time computation. An algorithm for efficient real-time operating system task scheduling has also been developed that minimizes energy consumption. Portable systems spend a lot of time in sleep mode. Idle power management strategies have been developed that consider the effect of leakage and duty-cycle on system lifetime. A hierarchical shutdown approach for systems characterized multiple sleep states has been proposed. Although the proposed techniques are quite general, their applicability and utility have been demonstrated using the MIT μAMPS wireless sensor node an example system wherever possible. To quantify software energy consumption, an estimation framework has been developed based on experiments on the StrongARM and Hitachi processors. The software energy profiling tool is available on-line. Finally, in energy constrained systems, we would like to have the ability to trade-off quality of service for extended battery life. A scalable approach to application development has been demonstrated that allows energy quality trade-offs.

Thesis Supervisor: Anantha Chandrakasan
Title: Associate Professor

Acknowledgements

I would like to express my sincere gratitude to Prof. Anantha Chandrakasan. I met him twice before I came to MIT and I was convinced that working with him would be a most stimulating and rewarding experience, which now in retrospect, seems to be one of the best decisions I ever made. Throughout my Master's and Ph.D. program, his ideas and suggestions, keen insight, sense of humor, never-say-die enthusiasm and perennial accessibility have been instrumental in getting this work done. Apart from academic counsel, I consider his professional accomplishments and career a model to emulate.

I would also like to thank Prof. John V. Guttag and Prof. Duane S. Boning for their time and effort in reviewing this thesis despite their incredibly busy schedules. Their inciteful comments and suggestions during committee meetings and draft reviews have been extremely helpful in adding a lot of fresh perspective in this work.

I would also like to thank all my colleagues in 38-107. Alice Wang for being the perfect effervescent colleague, sharing her MIT expertise, the foodtruck lunches, and always being there to help. Thanks to her I managed to sail across the rough waters of 6.374 as a TA. Manish Bhardwaj and Rex Min for burning midnight oil with me, not to forget the numerous wagers that were thrown in the wee hours of the morning! During these nocturnal discussions the meaning of life and the cosmic truths were discovered many times over! Jim Goodman for his unbelievable patience in answering all my questions about the StrongARM and Framemaker. Special thanks to Gangadhar Konduri for helping me get settled when I first came to MIT. His widespread *desi* network got my social life going. Wendi Heinzelman, for exemplifying the benefits of focus in getting your Ph.D. SeongHwan Cho, for his reassuring presence and for making TAing 6.374 so much less grungy. Travis Furrer for his outstanding foundation work with the eCos operating system and his perpetual keenness to help whether it was the idiosyncrasies of Java or the installation of a painful Solaris patch. Nathan Ickes for getting me up to speed with his processor board.

Eugene Shih, for having answers to all my IT questions! James Kao, Frank Honore, Theodoros Konstantakopoulos, Raul Blazquez-Fernandez, Benton Calhoun, Paul Peter, Scott Meninger, Joshua Bretz, and Oscar Mur-Miranda for being such vivacious and helpful officemates. I'd also like to take this opportunity to thank the veterans - Raj, Duke, Vadim and Tom Simon for helping me despite their tapeout deadlines and teaching me how to take a more critical and less gullible view of things. I wish to thank all my 'non-EECS' friends at MIT for making the home-on-the-dome work out for me. Together we roller coasted the ups and downs of graduate life and refused to drink water from the fire hose!

Finally, I'd like to thank the people who are dearest to me and without who's love and support none of this would have been possible. My wonderful wife Deepali, for her infinite patience, love and encouragement. My mother, Anjali Sinha, my father R. N. Sinha, and my sister, Neha, to whom I owe everything. Their blessing, guidance, love and sacrifice has made me what I am today.

Amit Sinha
August, 2001

Table of Contents

1 Introduction **17**

1.1 Energy Efficient System Design....17
1.1.1 Portable Systems....17
1.1.2 Processor versus Battery Technology....18
1.1.3 Power Efficiency in Desktops and Servers....19
1.1.4 Reliability and Environmental Cost....21
1.1.5 Thesis Scope....22

1.2 Avenues for Energy Efficiency....22
1.2.1 Sources of Power Consumption....22
1.2.2 Low Power Circuit Design....23

1.3 Software versus Hardware Optimizations....24
1.3.1 Low Power Microprocessors....24
1.3.2 Advantages of a Software Approach....26

1.4 Thesis Overview....27
1.4.1 Related Work....28
1.4.2 The MIT mAMPS Project: An Application Driver....29
1.4.3 Thesis Organization....30

2 Active Power Management **31**

2.1 Variable Voltage Processing....32
2.1.1 Previous Work....32
2.1.2 Workload Prediction....33
2.1.3 Energy Workload Model....34
2.1.4 Variable Power Supply....35

2.2 Workload Prediction....36
2.2.1 System Model....36
2.2.2 Frequency and Minimum Operating Voltage....37
2.2.3 Markov Processes....38
2.2.4 Prediction Algorithm....39
2.2.5 Type of Filter....40

2.3 Energy Performance Trade-offs....43
2.3.1 Performance Hit Function....43
2.3.2 Optimizing Update Time and Taps....46

2.4 Results....47

2.5 Summary of Contributions....48

3 Power Management in Real-Time Systems **51**

3.1 Aperiodic Task Scheduling....52
3.1.1 Performance Evaluation Metrics in Real-Time Systems....52
3.1.2 The Earliest Deadline First Algorithm....53

3.1.3 Real-Time Systems with Variable Processing Rate 54
3.1.4 The Slacked Earliest Deadline First (SEDF) Algorithm 55
3.1.5 Results 58
3.1.6 Upper Bound on Energy Savings 61
3.2 Periodic Task Scheduling 62
3.2.1 Dynamic Priority Assignments 62
3.2.2 Static Priority Assignments 63
3.3 Summary of Contributions 64

4 Idle Power Management 65

4.1 Previous Work 65
4.2 Multiple Shutdown States 66
4.2.1 Advanced Configuration and Power Management Interface 67
4.3 Sensor System Models 68
4.3.1 Sensor Network and Node Model 68
4.3.2 Power Aware Sensor Node Model 69
4.3.3 Event Generation Model 71
4.4 Shutdown Policy 72
4.4.1 Sleep State Transition Policy 72
4.4.2 Missed Events 74
4.5 Java Based Event Driven Shutdown Simulator 76
4.5.1 Results 77
4.6 Summary of Contributions 80

5 System Implementation 83

5.1 Embedded Operating Systems Overview 84
5.1.1 Windows CE 84
5.1.2 Palm OS 85
5.1.3 Redhat eCos 86
5.2 Sensor Hardware 87
5.2.1 DVS Circuit 88
5.2.2 Idle Power Management Hooks 92
5.2.3 Processor Power Modes 93
5.3 OS Architecture 95
5.3.1 Kernel Overview 96
5.3.2 Application Programming Interface 97
5.3.3 Web Based Application Development Tool 99
5.4 System Level Power Management Results 101
5.4.1 Active Mode Power Savings 101
5.4.2 Idle Mode Power Savings 102
5.4.3 Energy Cost of Operating System Kernel Functions 104
5.5 Summary of Contributions 106

6 Software Energy Measurement **107**
6.1 Factors Affecting Software Energy 107
6.2 Previous Work 108
6.3 Proposed Methodology 110
6.3.1 Experimental Setup 110
6.3.2 Instruction Current Profiles 111
6.3.3 First Order Model 114
6.3.4 Second Order Model 115
6.4 Leakage Energy Measurement 118
6.4.1 Principle 118
6.4.2 Observations 119
6.4.3 Explanation of Exponential Behavior 122
6.4.4 Separation of Current Components 124
6.4.5 Energy Trade-Offs 125
6.5 JouleTrack 128
6.6 Summary of Contributions 129

7 Energy Scalable Software **131**
7.1 Energy Scalability Example 132
7.2 Formal Notions for Scalability 134
7.3 Energy Scalable Transformations 135
7.3.1 Filtering Application 135
7.3.2 Image Decoding Application 138
7.3.3 Classification using Beamforming 141
7.4 Energy Scalability with Parallelism - Pentium Example 144
7.5 Summary of Contributions 146

8 Conclusions **147**
8.1 Contributions of this Thesis 148
8.2 Future Work 150

References **153**

A Operating System Energy Characterization **161**

B Current Measurements **167**

C mAMPS Webtool **171**

D JouleTrack Screenshots **173**

E Scalable Image Decoding **175**

List of Figures

Figure 1-1: The portable electronic products market [4] 17

Figure 1-2: Battery technologies 18

Figure 1-3: Nationwide electricity consumption attributed to office and network equipment showing possible power savings using current technology [5] 20

Figure 1-4: Microprocessor power consumption trends 20

Figure 1-5: Energy efficient system design and scope of the thesis 27

Figure 2-1: Dynamic voltage and frequency scaling 32

Figure 2-2: 60 second workload trace for three processors 34

Figure 2-3: (a) Energy vs. workload, (b) Typical DC/DC converter efficiency profile, and, (c) Current vs. workload 36

Figure 2-4: Block diagram of a DVS processor system 37

Figure 2-5: Frequency and minimum operating voltage (normalized plot) 38

Figure 2-6: Prediction performance of the different filters 42

Figure 2-7: Workload tracking by the LMS filter 42

Figure 2-8: Performance hit, settling time notions 43

Figure 2-9: Average and maximum performance hits 45

Figure 2-10: Energy consumption (normalized to the no DVS case) as a function of update time and prediction filter taps 46

Figure 2-11: Effect of number of discrete processing levels, L 48

Figure 3-1: Greedy scheduling of processing rate 54

Figure 3-2: Illustrating the parameters involved in SEDF 56

Figure 3-3: Completion probability, weighted energy and optimum processing rate 58

Figure 3-4: Optimum processing rate as a function of processor utilization and slack 58

Figure 3-5: Example of EDF and SEDF scheduling 59

Figure 3-6: Comparing the performance of EDF and SEDF 60

Figure 3-7: Energy ratio as a function of processor utilization 60

Figure 4-1: ACPI interface specification on the PC 67

Figure 4-2: Sensor network and node architecture 69

Figure 4-3: State transition latency and power 73

Figure 4-4: Transition algorithm to 'almost off' s4 state 76
Figure 4-5: Java based event driven sensor network shutdown simulator 77
Figure 4-6: Java based event driven sensor network shutdown simulator 78
Figure 4-7: (a) Spatial distribution of events (Gaussian) and (b) Spatial energy consumption in the sensor nodes 79
Figure 4-8: Event arrival rates at a node and corresponding limits on state utilization 80
Figure 4-9: Fraction of events missed versus energy consumption 80
Figure 5-1: mAMPS processor board (designer: Nathan Ickes) 88
Figure 5-2: DVS circuit schematic 89
Figure 5-3: Sequence of operations during a voltage and frequency switch 91
Figure 5-4: Idle power management hooks on the sensor processor board 93
Figure 5-5: eCos architecture showing the power management layer that was added 95
Figure 5-6: Application development toolchain for the mAMPS sensor node 100
Figure 5-7: System level power savings from active power management using DVS 101
Figure 5-8: Degradation in DVS savings with increase in workload variance 102
Figure 5-9: System level power savings distribution 103
Figure 5-10: Battery life improvement in the sensor node compared to a node with no power management as a funtion of duty cycle and active workload 104
Figure 5-11: Average, maximum and minimum energy consumption of kernel function calls (grouped by category) 105
Figure 6-1: Experimental setup for instruction/program current measurement 109
Figure 6-2: StrongARM SA-1100 experimental setup 111
Figure 6-3: (a) Strong SA-1100 average instruction set current consumption (b) Instruction set current distribution 112
Figure 6-4: Distribution of power consumption in the ARM core [79] 113
Figure 6-5: Hitachi SH-4 average instruction set current consumption 113
Figure 6-6: Current variations within instructions on the StrongARM 114
Figure 6-7: Program current consumption as a function of operating point 115
Figure 6-8: Average current and supply voltage at each operating frequency of the StrongARM SA-1100 117
Figure 6-9: First and second order model prediction errors 118

Figure 6-10: FFT energy consumption 120
Figure 6-11: FFT charge consumption 121
Figure 6-12: Leakage current variation 122
Figure 6-13: Effect of transistor stacking 123
Figure 6-14: Static, dynamic and total current 123
Figure 6-15: Leakage current fraction 125
Figure 6-16: FFT energy components 126
Figure 6-17: Low duty cycle effects 127
Figure 6-18: JouleTrack block diagram 128
Figure 6-19: Timing estimation engine within Jouletrack 129
Figure 7-1: E-Q performance of power series algorithm 133
Figure 7-2: E-Q formal notions 135
Figure 7-3: FIR filtering with coefficient reordering 136
Figure 7-4: E-Q graph for original and transformed FIR filtering 137
Figure 7-5: Energy inefficiency of unsorted system compared to the sorted case 138
Figure 7-6: 8x8 DCT coefficient magnitudes averaged over a sample image 140
Figure 7-7: E-Q graph for FM-IDCT vs normal IDCT 140
Figure 7-8: Illustrating the incremental refinement property with respect to computational energy of the FM-IDCT algorithm 140
Figure 7-9: Beamforming for data aggregation 141
Figure 7-10: Sensor testbed 142
Figure 7-11: E-Q snapshot for Scenario 1 143
Figure 7-12: "Sort by significance" preprocessing 143
Figure 7-13: E-Q snapshot for Scenario 2 144
Figure 7-14: Pentium III SIMD registers and data type 145
Figure C-1: mAMPS operating system and software webtool 171
Figure D-1: JouleTrack remote file upload and operating point selection 173
Figure D-2: JouleTrack energy profiling output 174

List of Tables

Table 2-1: DVS energy savings ratio (Eno-dvs/Edvs) [N = 3, T = 5 s] 47
Table 3-1: Real-time performance metrics 53
Table 4-1: Useful sleep states for the sensor node 70
Table 4-2: Sleep state power, latency and threshold 74
Table 5-1: SA-1110 core clock configurations and minimum core supply voltage 90
Table 5-2: Power consumption in various modes of the SA-1110 94
Table 5-3: Native kernel C language API 98
Table 5-4: Primitive power management functions 99
Table 5-5: Measured power consumption in various modes of the sensor 102
Table 6-1: Weighting factors for K = 4 on the StrongARM 118
Table 6-2: Leakage current measurements 124
Table 7-1: Power series computation 133
Table 7-2: Power savings from parallel computation 146
Table A.1: Energy characterization of various OS function calls 162
Table B.1: SA-1100 Instruction Current Consumption 167
Table B.2: SH7750 Instruction Current Consumption 168
Table B.3: Program Currents on SA-1100 169
Table E.1: Scalability in Image Decoding 175

Chapter 1

Introduction

Energy efficient system design is becoming increasingly important with the proliferation of portable, battery-operated appliances such as laptops, Personal Digital Assistants (PDAs), cellular phones, MP3 players, etc. Saving energy is becoming equally important in servers and networking equipment as their perennially increasing numbers are resulting in increasing electricity and cooling costs.

1.1 Energy Efficient System Design

1.1.1 Portable Systems

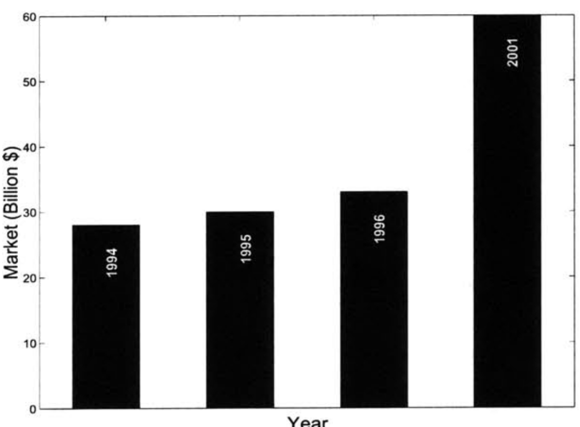


Figure 1-1: The portable electronic products market [4]

Figure 1-1 shows the trends in the portable electronic products market over the past few years. While traditional forms of mobile computing will continue to rise, analysts project that the evolution of wireless Personal Area Networks (PANs), with enabling technologies such as Bluetooth [1] becoming standardized, and third generation cellular services [2] that will enable wireless internet access and multimedia content delivery over cellular phones, there will be an exponential increase in the portable electronics market.

Another embedded application domain that is emerging is wireless networking of sensors for distributed data gathering [3].

1.1.2 Processor versus Battery Technology

One of the most important design metrics in all portable systems is low energy consumption. Energy consumption dictates the battery lifetime of a portable system. People dislike replacing or re-charging their batteries frequently. They also do not wish to carry heavy batteries with their sleek gadgets. As such, the energy constraints on portable devices are becoming increasingly tight as complexity and performance requirements continue to be pushed by user demand. Incredible computational power is being packed into mobile devices these days. Processor speeds have doubled approximately every 18 months as predicted by Moore's law [6]. There has also been a corresponding increase in power consumption in processors. In fact, microprocessor power consumption has gone up from under 1 Watt to over 50 Watts over the last 20 years.

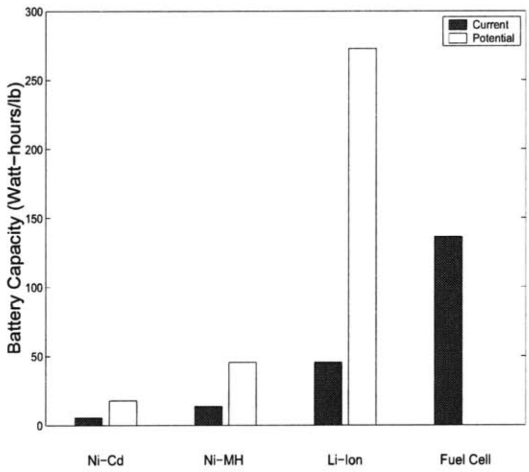


Figure 1-2: Battery technologies

While processor speed and power consumption have increased rapidly, the corresponding improvement in battery technology has been slow. In fact, battery capacity has increased by a factor of less than four in the last three decades [7]. Figure 1-2 shows the current state-of-art in battery technology. In [4] it has been speculated that battery technology is fast approaching the limits set by chemistry. Although newer technologies promise higher battery capacities, all of them have their share of problems. Nickel-Metal Hydride

(Ni-MH) although lighter than Ni-Cd, have increased recharge time. Lithium-Ion batteries promise high energy density, higher number of recharge cycles, little memory effect and low self-discharge rate (longer shelf life between recharging). However, they are higher priced, require protective circuitry to assure safe use and offer limited discharge rates. Other technologies such as Lithium Polymer and Methanol Fuel Cells are still in their experimental stages. For a detailed analysis of battery technology, the reader is referred to [4]. The bottom line is that we can expect only incremental improvements in battery technology while power consumption will rise much faster. Under these circumstances, energy efficient system design is becoming indispensable.

1.1.3 Power Efficiency in Desktops and Servers

While the number of portable gadgets has increased significantly, the corresponding increase in desktop and office equipment too has been very steady, albeit less dramatic. While energy consumption is the important design metric in portable systems, power consumption is the appropriate design metric for desktops and servers[1]. An increased awareness towards low power consumption translates to a significant reduction of electricity costs as well as cost reduction in cooling and packaging power hungry processors.

A recent government survey showed that the cumulative electricity consumption of all office and network equipment is the US was almost 100 TWHr/year [5]! (For comparison, the *total* residential electricity consumption in the US was about 1100 TWHr/year in the year 2000). Many organizations have expressed concern over the rising electricity consumption attributed to the internet and to commercial desktop computation. In the same report [5], it has been estimated that Power Management (PM) by saturated use of the limited energy saving mechanisms available in today's office equipment can result in about 35% reduction in power consumption as shown in Figure 1-3. Figure 1-4 shows the dra-

1. There is a significant difference between energy and power. Energy is the product of average power consumed and the time over which it occurred. For portable systems, energy efficiency is more important. Power, for example, can be reduced by simply slowing down the processor but that will not improve energy efficiency of the task since the execution time will increase proportionately. In this thesis, power and energy have been used interchangeably. However, our ultimate goal is energy efficiency.

matic increase in general-purpose processor (which go into servers and desktops) power consumption over the last few years.

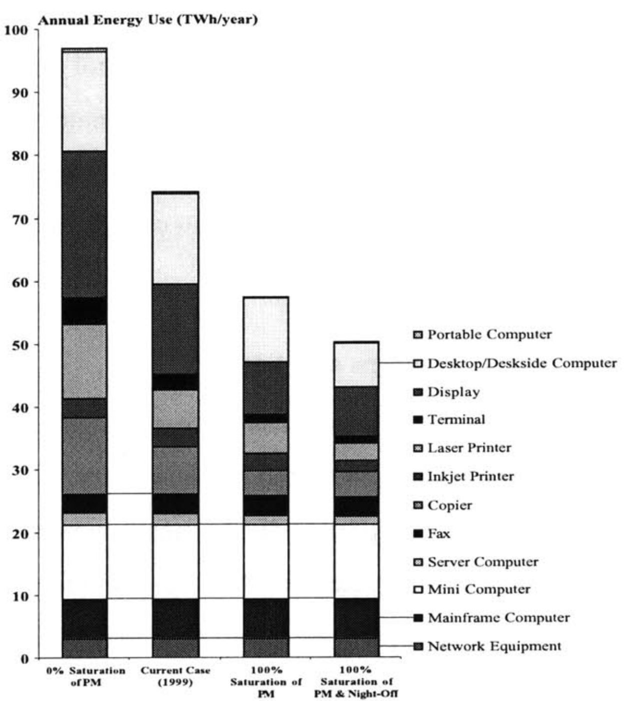


Figure 1-3: Nationwide electricity consumption attributed to office and network equipment showing possible power savings using current technology [5]

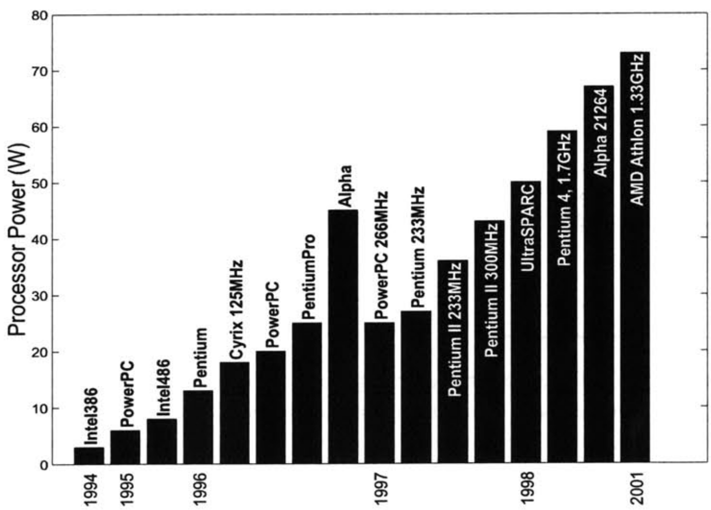


Figure 1-4: Microprocessor power consumption trends

1.1.4 Reliability and Environmental Cost

While electricity is the most tangible power cost in servers and network equipment, power consumption in these processors also affects cost in other ways.

- *Heat Dissipation* - Power in digital circuits is dissipated in the form of heat. The generated heat has to be dissipated for the device to continue to function. If the temperature of the integrated circuit rises beyond the specified rating, the operating behavior of circuits could change. In addition, the likelihood of catastrophic failure (through mechanisms such as electromigration, junction fatigue, gate oxide breakdown, thermal runaway, package meltdown, etc.) also increases exponentially. It has been shown that component failure rate doubles for every $10^{o}C$ rise in operating temperature. Today, die surface temperatures are already well over $100^{o}C$ and sophisticated packaging and cooling mechanisms have been deployed [8]. This increases system cost substantially. For example, a processor with less than 1 W power consumption requires a plastic package costing about 1 cent per pin. If the power consumption is over 2 W, ceramic packages costing about 5 cents per pin would be required. Therefore, low power design translates to lower system cost and increased reliability. Lower power consumption also means reduced air-conditioning costs which is also significant.
- *Environmental Factors* - A typical university like MIT might have about 10,000 computers. An average computer dissipates about 150 W of power. This translates to 3 million KWh per annum (costing $240,000) of electricity consumption assuming only business hours of operation. The equivalent greenhouse gas emission is about 2250 tons of carbon dioxide and 208,000 trees[1] are required to offset this! With the proliferation of digital systems, their environmental impact will only aggravate if low power design methodologies are not incorporated. The use of smart operating systems that shut off idle portions of a system can have a dramatic impact on average power consumption. Similarly, smart software can reduce system energy consumption by utilizing resources optimally.

---

1. An average tree absorbs about 22 lbs of carbon dioxide a year

1.1.5 Thesis Scope

This dissertation is an exploration of software techniques for energy efficient computation. We have proposed and demonstrated software strategies that significantly improve the energy efficiency of digital systems by exploiting software controlled active and idle power management. Algorithmic techniques for energy efficient computation have also been demonstrated.

1.2 Avenues for Energy Efficiency

1.2.1 Sources of Power Consumption

Power dissipation in digital circuits can be classified into two broad categories. The most prominent source of power dissipation is capacitive switching which results from the charging and discharging of the output of a CMOS gate. The switching power consumption of a CMOS circuit can be represented as [9]

$$P_{switch} = \alpha C_L V_{dd}^2 f \tag{1-1}$$

where C_L is the average switched capacitance per clock cycle, V_{dd} is the power supply voltage, f is the operating frequency and α is the activity factor of the circuit.

The other source of power consumption that is becoming significant is leakage. Leakage is a static power consumption mechanism and primarily results from sub-threshold transistor current [9]. The sub-threshold leakage current in a transistor depends exponentially on how close the gate voltage is to the transistor threshold. For example, reducing the threshold voltage from 0.5 V to 0.35 V can result in sub-threshold leakage increasing by a factor of over 20. The closer the gate voltage to the threshold, the higher the leakage, since the device gets partially on and starts operating in a bipolar mode. As operating voltages and thresholds are reduced, leakage power consumption is becoming increasingly important. Leakage currents were approximately 10-20 pA/μm with threshold voltages of 0.7 V, whereas today, with threshold voltages of 0.2-0.3 V they can be as much as 10-20 nA/μm.

1.2.2 Low Power Circuit Design

There is a wealth of research done on low power circuit design methodologies at all levels of the system abstraction. The primary focus has been on reducing the switched capacitance and lowering the supply voltage. Lowering switched capacitance results in linear reduction in power consumption. An example of a technique that is used commonly to reduce switched capacitance in microprocessors is clock gating. Clock gating shuts off the clock to portions of the processor that are not currently in use. This avoids unnecessary transition activity and reduces switched capacitance [10]. A more aggressive technique is to power down unused portions of the circuit.

Power can be traded off for operating speed. Simply reducing the operating frequency can result in linear reduction in power consumption at the cost of performance. Substantially higher savings can result from reducing the operating voltage as well[1]. Reduced operating voltage is probably the most effective technique for low power. For example, driving long on-chip interconnects on integrated circuits dissipates a lot of power. Reduced voltage swing bus driver circuits are employed to reduce power on long interconnects [11]. Silicon area can also be traded off for power. A classic example of this technique is parallelism. By duplicating hardware and reducing the operating frequency and voltage, throughput can be kept constant at a lower power dissipation [10]. Another interesting technique involves energy-recovery CMOS circuits using adiabatic logic. The basic idea here is that by controlling the length and shape of signal transitions between logic levels, the expended energy can be asymptotically reduced to an arbitrary small degree [12]. However, such a scheme has practically limitations.

With leakage currents becoming a substantial portion of the power budget in contemporary microprocessors, several leakage reduction mechanisms have been proposed. The use of multiple-threshold CMOS (MT-CMOS) where low threshold (i.e., fast) devices are placed in the critical path and high threshold (i.e., slower) devices are placed in non-critical paths has been used effectively to counter leakage [13][14]. Substrate biasing can also

---

1. There exists an almost linear relationship between minimum operating voltage required and corresponding operating frequency. Obviously, from a low power standpoint it pays off to work at the lowest possible operating voltage.

be used to actively vary device threshold. For a detailed overview of various low power circuit techniques the reader is referred to [15].

1.3 Software versus Hardware Optimizations

It has been shown in separate applications that dedicated hardware implementations can out perform general purpose microprocessors/DSPs by several orders of magnitude in terms of energy consumption [23][24]. However, dedicated implementations are not always feasible. Application Specific Integrated Circuits (ASICs) are getting increasingly expensive to design and manufacture and are a solution only when speed constraints dictate otherwise. Furthermore, introducing revisions and changes into hardwired solutions is expensive and time-consuming. The breaking of the $5 threshold for 32-bit processors has resulted in an explosion in the use of general purpose microprocessors and DSPs in high-volume embedded applications [25]. In addition, the power efficiency gap between dedicated ASICs and their programmable counterparts is reducing with the introduction of various low power processors some of which are described in the next section.

1.3.1 Low Power Microprocessors

As the demand for portable electronics has increased, several low power processors have entered the market. These processors consume one to two orders of magnitude lower power than some of the contemporary microprocessors listed in Figure 1-4. Most of the power savings in these processors comes from three sources - (i) Smart circuit design, using techniques mentioned in [15], (ii) Throwing away lesser used functionality, i.e., architectural trimming, and, (iii) Voltage scaling and clock gating. Some of the more prominent processors are as follows:

- *StrongARM* - Built on the ARM architecture, the Intel StrongARM family [16] of processors delivers a combination of high performance and low power consumption with features that can handle applications such as handheld PCs, smart phones, web phones, etc. The StrongARM SA-1100 processor, for example, has a peak performance of 206 MHz while consuming only 350 mW of power! Most of the power reduction over a high-performance processor like a Pentium is obtained by throwing away power hungry functional blocks like floating point units, reducing cache sizes

and simplifying the unnecessarily complex x86 ISA [17]. Floating point computation is emulated in software. Aggressive clock gating along with an efficient clock distribution strategy is employed for further power reduction.

- *Crusoe* - Transmeta's Crusoe family of processors has specifically been designed for low power applications [18]. The processor features the LongRun technology which allows the processor to run at a lower frequency and operating voltage (and therefore reduced power consumption) during periods of reduced processor load. The TM5400, for example, can scale from 500 MHz at 1.2 V to 700 MHz at 1.6 V. The Crusoe architecture is a flexible and efficient hardware-software hybrid that replaces millions of power-hungry transistors with software, while maintaining full x86 compatibility. At the heart of Crusoe lies an effective code morphing technique [19] that dynamically translates complex x86 instructions into the internal VLIW[1] instructions of Crusoe while fully exploiting run-time statistics to improve performance and reduce power consumption.
- *SuperH* - The Hitachi SuperH (SH) family of processors is another alternative available as a low power platform [20]. Hitachi designed these families in low-power submicron CMOS processes with low-voltage capabilities. Low static operating current is stressed in all circuit designs and low dynamic (peak) currents are guaranteed by logic and circuit design. All implementations include a variety of software-controlled power reduction mechanisms. Each family embodies a selection from a palette including standby and sleep modes, clock speed control and selective module shutdown. For example, the SH-3 permits the clocks for the CPU, the on-chip peripherals and the external bus to be separately optimized. This flexibility permits the system designer to choose the optimum combination of low power and system responsiveness for each application.
- *DSPs* - Digital Signal Processors can deliver a better performance to power ratio for computationally intensive operations. DSPs differ from general purpose microprocessors in that they have narrow data widths, high speed multiply-accumulate, multi-

1. Very Large Instruction Word - An architecture where several RISC instructions, which can be executed in parallel, are packed into one long instruction (usually by the compiler). VLIW CPUs have more functional units and registers than CISC or RISC CPU but do not need instruction reordering and branch prediction units.

ple memory ports with specialized memory addressing, zero overhead loops and repeat instructions. Among DSPs themselves several lower power versions exist. Prominent among them are the TMS320C5xx family of DSPs from Texas Instruments [21] and the StarCore family [22].

1.3.2 Advantages of a Software Approach

While it is true that maximum power savings are possible through hardware optimizations, the introduction of low power processors as discussed in the previous section coupled with the following benefits, makes a software solution the preferred approach:

- *Flexibility* - One of the most important considerations that has encouraged software solutions is flexibility. Protocols and standards are constantly evolving and new standards are being incorporated every day. For example, the MPEG video standard started off with MPEG-1 and MPEG-2 and the MPEG committee is now working on MPEG-7 [26]. New radio standards such as Bluetooth have evolved along with protocols such as the Wireless Application Protocol (WAP) [27], and while they are far from being fully implemented, revisions are already in progress. While most standards and protocols support backward compatibility, market and customer pressures make upgrades a necessity. A software solution allows the flexibility of a field upgrade. Users can download the modified patches from the internet while preserving their investment and getting better services. Software also offers fast prototyping solutions for evolving technologies on a mature time-tested hardware platform.
- *Time-to-Market* - With technology evolving at such a rapid pace, time-to-market for a product is everything. The design and testing time required for a moderately complex ASIC can run well over a year with today's Computer Aided Design (CAD) environments. Although such a product might be an order of magnitude more efficient than a software solution on a standard platform, the design latency involved can render the product obsolete by the time it hits the market. On the other hand, the presence of powerful and mature software development environments along with an abundance of skilled manpower gives software shorter and more flexible design cycles. This coupled with economics involved in having programmable solutions on general purpose processors rather than hardwired ones, have engendered a shift

towards programmable solutions.

1.4 Thesis Overview

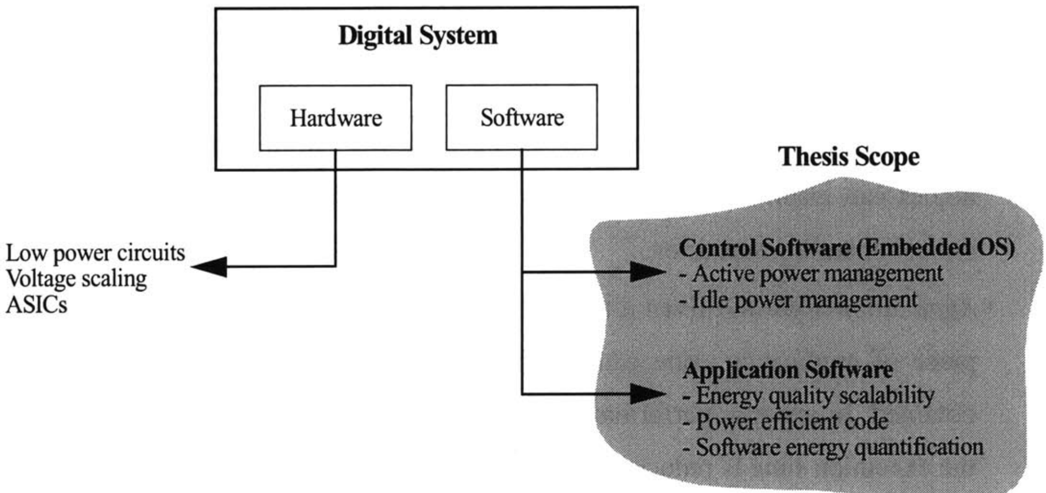


Figure 1-5: Energy efficient system design and scope of the thesis

Designing a complex digital system is non-trivial, often involving an intricate inter-dependent development of hardware and software. Figure 1-5 shows the various aspects of a digital system that can be optimized for energy efficiency. As discussed in the previous section, dedicated hardware implementations can yield substantial improvements in power consumption but their cost and development time might be prohibitive. Instead, most energy conscious commercial digital systems utilize some standardized low power hardware platform and custom software for implementation.

Several researchers have investigated techniques for low power implementation of microprocessors and DSPs and a good summary of these techniques can be found in [15]. Most of this research has focussed on circuit and hardware techniques. *This work investigates avenues open in software for energy efficient system design*. The contributions made in this thesis can be broadly characterized into two categories:

- *Control Software*: We refer to the control software as the Operating System (OS). The primary function of the OS is control, e.g., allocation of resources such as memory, servicing interrupts, scheduling applications/tasks, etc. The following OS techniques for system energy efficiency have been developed and implemented in this

thesis: (i) Active Power Management, where the OS provides just the right amount of power required to run the system at the desired performance level by adaptive control of voltage and frequency. Optimum scheduling algorithms for energy efficient real-time computing have also been proposed. (ii) Idle Power Management, where the OS puts portions of the system into multiple low power sleep states and wakes them up based on processing requirement. Smart shutdown algorithms have been proposed and demonstrated in the thesis. It has been shown that utilizing the proposed techniques can result in 1-2 orders of magnitude reduction in energy consumption for typical operating scenarios.

- *Application Software*: Even if the hardware and OS are designed to be efficient, a bad piece of application code can reduce any energy benefits that would have been obtained. In general, performance optimized software is also energy efficient since the execution time is reduced. In this thesis, other avenues for improved application software energy efficiency have been explored. Techniques to improve the Energy-Quality scalability of software wherein the application can trade-off quality of service for lower energy consumption have been proposed and demonstrated for a variety of applications. Fast software energy estimation tools have been developed to quantify the energy consumption of a piece of application code.

1.4.1 Related Work

Software energy efficiency is a relatively unexplored area of research. The idea of workload dependent processing for energy efficiency in an ASIC was demonstrated in [31]. Implementing such techniques in general purpose processors poses both circuit and software challenges. The operating system has been traditionally used for resource management but not necessarily for energy management. We have demonstrated a performance on demand approach for computation using operating system scheduling and smart workload prediction on general purpose processors. In addition, we have proved the optimality of our scheduling algorithm. Event driven computation has been used for a long time. The idea of turning off devices when not it use is a well-known strategy for saving energy. Predictive system shutdown techniques have been explored in [57]. Dynamic power management strategies have been proposed in [58] and related works by the same

author. In this thesis we have proposed the use of *multiple* shutdown states. We have shown that such granularity gives significantly better energy scalability in the system. The proposed scheme accounts for transition latencies and event statistics in a formal way. Our results indicate that an order of magnitude energy savings can be expected from using our techniques.

Instruction level power analysis of software was first proposed in [75]. This methodology is cumbersome and error prone. We have demonstrated a software energy estimation methodology which is an order of magnitude faster with lower estimation error than that proposed in [75]. Our estimation tool is available online [73]. We have also outlined a technique to estimate the leakage energy consumption at the software level.

Incremental refinement in algorithms has been studied in [82]. We have demonstrated algorithmic transformations that improve the energy scalability of an algorithm by improving the incremental refinement property in the context of energy consumption.

1.4.2 The MIT μAMPS Project: An Application Driver

Over the past few years, the design of micropower wireless sensor systems has gained increasing importance for a variety of commercial and military applications ranging from security devices and medical monitoring to machine diagnosis and chemical/biological detection. Networks of microsensors (vs. a limited number of macrosensors) can greatly improve environment monitoring and provide significant fault tolerance. Significant research has been done on the development of low-power Micro Electro Mechanical System (MEMS) [29] sensors that could be embedded onto the substrate. We assume that the basic sensing technology is available. The goal of the μAMPS Project [28] is to develop a framework for implementing adaptive energy-aware distributed microsensors. As such, programmability is a key requirement and energy efficient protocols, algorithms and software implementation strategies are crucial.

The μAMPS system has all the attributes of an energy constrained system and will be used as an application driver, wherever possible, to demonstrate the feasibility of a proposed energy efficient solution in this thesis. The sensor nodes are expected to have battery lifetimes of approximately a year. With the current battery capacity we can only expect them to last a few weeks at most. A smart operating system on the sensor node can

substantially improve the energy efficiency using active and idle power management and such savings are quantified in the thesis. Sensing and data processing algorithms running on these sensor nodes have been designed to demonstrate the concept of Energy-Quality scalability.

1.4.3 Thesis Organization

This thesis is organized as follows. Chapters 2 and 3 describe our proposed operating system directed active power management methodology. A rigorous analytical framework for real-time and non real-time operating systems has been developed. In Chapter 4, the multiple sleep state based shutdown scheme is described. The use of multiple sleep states has been shown to improve the energy scaling granularity and simulation results are included to support our claim. Chapter 5 describes the system level energy savings that were obtained on the μAMPS sensor node by exploiting the active and idle power management techniques that have been proposed. It discusses the sensor hardware as well as the operating system that was developed to enable power management on the node. The overhead of the operating system itself is also quantified along with the expected battery life improvement. Chapter 6 outlines the software energy estimation methodology that we have developed. Our leakage estimation technique is also described here. The architecture of the web-based software energy estimation tool is also outlined. Chapter 7 describes our proposed algorithmic approach for energy scalable software using algorithmic transformations and parallelism hooks available in processors. Finally, the contributions and conclusions drawn from this thesis are summarized in Chapter 8.

Chapter 2

Active Power Management

A system can be in an active or idle mode. The operating system can be used to manage active power consumption in an energy constrained system. For example, when a user is running a spreadsheet application on his laptop, the processor utilization profile in the laptop is characterized by intermittent peaks when the user saves/updates the spreadsheet. The operating system can intelligently reduce the performance of the processor (by reducing the operating frequency and voltage) to the level required by the application(s) such that there is no visible loss in observed performance while the energy consumption is reduced in accordance with Equation 1-1. At present only a very few processors have dynamic frequency control. Some of these processors were discussed in Section 1.3.1 (StrongARM SA-1100 [16] and Transmeta's Crusoe Processor [18]). The Intel Pentium III features a very primitive frequency control technology called SpeedStep which allows a laptop to run at a lower frequency when running off a battery supply. However, no frequency change is allowed at runtime (e.g., if the user plugs in the power supply, performance cannot be boosted without re-booting). The StrongARM-2 processor, with built in frequency and voltage control, it at present the most promising dynamic voltage and frequency processor.

As dynamic voltage and frequency processors become increasingly available, operating systems will have to be designed to exploit this feature to maximize energy efficiency. In this chapter, operating system directed power management using an adaptive performance scheme is explored. Adaptive performance is enabled by a dynamic variation of operating voltage and frequency of the processor. To make the performance loss invisible to the user, the scheduling of operating voltage and frequency has to be done based on the workload profile of the processor. In order to effectively speculate on the workload of the system a prediction strategy is presented that employs an adaptive workload filtering scheme [30]. The effects of update frequency and filtering strategy on the energy savings is analyzed. A performance hit metric is defined and techniques to minimize energy under

a given performance requirement are outlined. Our results demonstrate that *energy savings by a factor of two to three* is possible with dynamic voltage and frequency scaling depending on workload statistics. Of course, if the workload is high all the time the energy savings will be lower. However, our measured data indicates that most processors in workstations and servers have low average utilization.

2.1 Variable Voltage Processing

2.1.1 Previous Work

Dynamic Voltage Scheduling (DVS) is a very effective technique for reducing CPU energy. Most systems are characterized by a time varying computational load. Simply reducing the operating frequency during periods of reduced activity results in linear decrease in power consumption but does not affect the total energy consumed per task as shown in Figure 2-1(a) (the shaded area represents energy). Reduced operating frequency implies that the operating voltage can also be reduced which results in quadratic energy reduction as shown in Figure 2-1(b). Significant energy benefits can be achieved by recognizing that peak performance is not always required and therefore the operating voltage *and* frequency of the processor can be dynamically adapted based on instantaneous processing requirement.

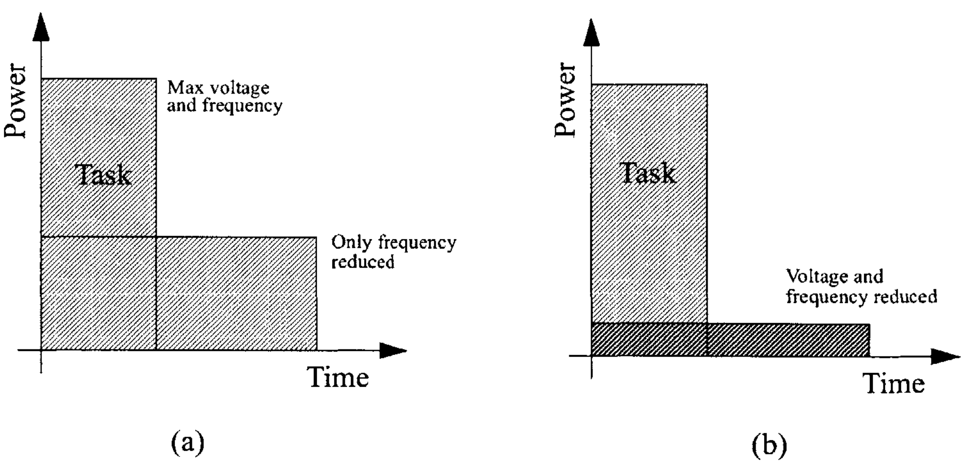


Figure 2-1: Dynamic voltage and frequency scaling

In [31] a low power DSP was designed with a variable power supply and it was shown that using adaptive dynamic voltage and frequency control substantial energy savings is

possible. The authors of [32] implemented a dynamic voltage and frequency microprocessor. Both these works have concentrated on circuit aspects of a variable voltage and frequency processor. In [33] various speed setting algorithms for variable frequency processors is analyzed, and it is shown that simple smoothing algorithms have a better performance than sophisticated prediction schemes. Our adaptive filter based prediction strategy is simple and effective. We have introduced the notion of a performance hit function and used it to optimize update rate and filter taps.

2.1.2 Workload Prediction

Figure 2-2 shows a 1 minute snapshot of the workload trace of three processors being used for three different types of applications: (i) a dialup server (characterized by numerous users logging in and out independently), (ii) a workstation (characterized by an single interactive user) and (iii) a UNIX file server (characterized by intermittent requests from the network). The varying workload requirements are at once apparent. We have used processor workload traces from these machines since such exhaustive data was not available from other embedded systems (e.g., sensors and laptops). The UNIX operating system provides a powerful API and tools to monitor and log various aspects of the processor. We were able to collect hours of data from these processors with minimal observational interference at different times of the day in an automated fashion. The data available from these processors has been used to test the efficacy of our proposed methodology.

The goal of DVS is to adapt the power supply and operating frequency to match the workload such that the visible performance loss is negligible. The crux of the problem lies in the fact that future workloads are often hard to predict. The rate at which DVS is done also has a significant bearing on performance and energy. A low update rate implies greater workload averaging which results in lower energy. The update energy and performance cost is also amortized over a larger time frame. On the other hand a low update rate also implies a greater performance hit since the system will not respond to a sudden increase in workload. While prior work has mostly focussed on circuit issues in dynamic voltage and frequency processors, we have proposed a workload prediction strategy based on adaptive filtering of the past workload profile. Several prediction schemes are ana-

lyzed. We also define a performance hit metric which is used to estimate the visible loss in performance and set update rate to keep the performance loss bounded.

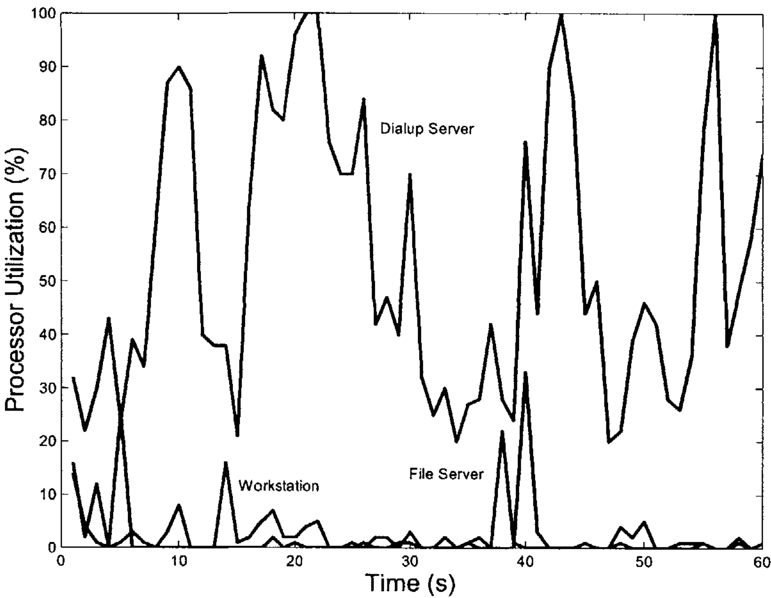


Figure 2-2: 60 second workload trace for three processors

2.1.3 Energy Workload Model

Using simple first order CMOS delay models it has been shown in [31] that the energy consumption per cycle is given by

$$E(r) = CV_0^2 T_s f_{ref} r \left[\frac{V_t}{V_0} + \frac{r}{2} + \sqrt{r\frac{V_t}{V_0} + \left(\frac{r}{2}\right)^2} \right]^2 \tag{2-1}$$

where C is the average switched capacitance per cycle, T_s is the period, f_{ref} is the operating frequency at V_{ref}, r is the normalized processing rate, i.e., $r = f / f_{ref}$ and $V_0 = (V_{ref} - V_t)^2 / V_{ref}$ with V_t being the threshold voltage. The normalized workload in a system is equivalent to the processor utilization. The operating system scheduler allocates a time-slice and resources to various processes based on their priorities and state. Often no process is ready to run and the processor simply idles. The normalized workload, w, over an interval is simply the ratio of the non-idle cycles to the total cycles, i.e., $w = (total_cycles - idle_cycles) / total_cycles$. The normalized processing rate is always in reference to the maximum pro-

cessing rate. In an ideal DVS system the processing rate is matched to the workload so that there are no idle cycles and utilization is maximum. Figure 2-3(a) shows the plot of normalized energy versus workload as described by Equation 2-1, for an ideal DVS system. Some important conclusions from the graph were derived in [31], (i) Averaging the workload and processing at the mean workload is more energy efficient because of the convexity of the $E(r)$ graph and Jensen's inequality [34]: $\overline{E(r)} \geq E(\bar{r})$. (ii) A small number of discrete processing rate levels (i.e., supply voltage, V_{dd}, and operating frequency, f) can give energy savings very close to the savings obtained from arbitrary precision DVS. This is because a few piecewise linear chords on the $E(r)$ graph can very closely approximate the continuous curve.

2.1.4 Variable Power Supply

A variable power supply can be generated using a DC/DC converter which takes a fixed supply and can generate a variable voltage output based on a pulse-width modulated signal[1]. It essentially consists of a power switch and a second order LC filter and is characterized by an efficiency that drops off as the load decreases, approximately as shown in Figure 2-3(b) [35]. At a lower current load, most of the power drawn from the supply gets dissipated in the switch and therefore the energy gains from DVS are proportionately reduced. Using a technique similar to the one used in the derivation of Equation 2-1, a first order current consumption equation can be expressed as

$$I(r) = I_{ref} r \frac{V_0}{V_{ref}} \left[\frac{V_t}{V_0} + \frac{r}{2} + \sqrt{r \frac{V_t}{V_0} + \left(\frac{r}{2}\right)^2} \right] \qquad (2\text{-}2)$$

where I_{ref} is the current drawn at V_{ref}. Using the DC/DC converter efficiency graph and the relative load current $I(r)$, we can predict the efficiency, $\eta(r)$. Figure 2-3(a) also shows the $E(r)$ curve after incorporating the efficiency of the DC/DC converter as shown in Figure 2-3(b) while Figure 2-3(c) shows the relative current consumption as a function of the workload (again assuming an ideal DVS system with $w = r$) as predicted by Equation 2-2. Efficient converter design strategies have been explored in [36].

1. For a circuit schematic please refer to Figure 5-2.

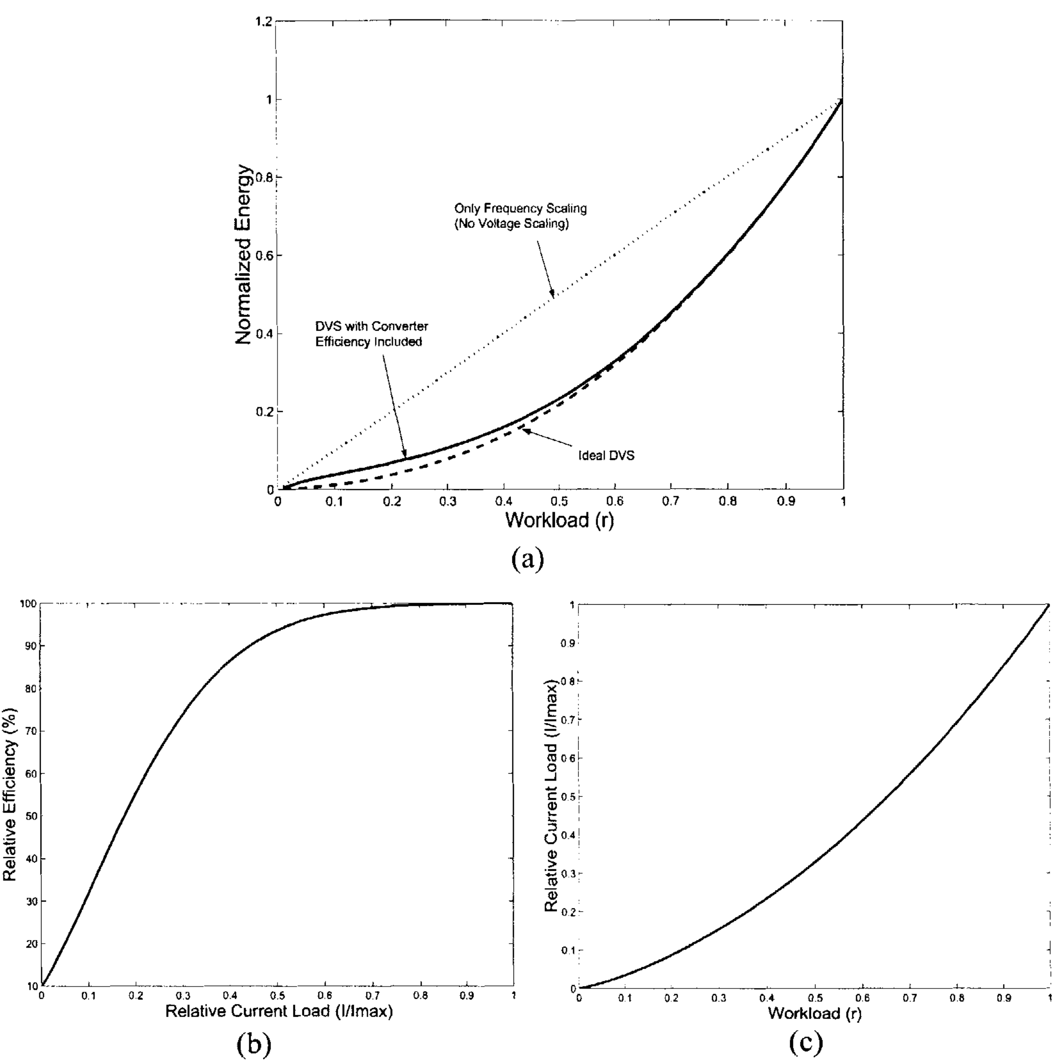


Figure 2-3: (a) Energy vs. workload, (b) Typical DC/DC converter efficiency profile, and, (c) Current vs. workload

2.2 Workload Prediction

2.2.1 System Model

Figure 2-4 shows a generic block diagram of the variable frequency and variable voltage processing system. The 'Task Queue' models the various event sources for the processor, e.g., I/O, disk drives, network links, internal interrupts, etc. Each of the n sources produces events at an average rate of λ_k, (k = 1, 2, .. , n). Typically a Poisson process is

assumed for such systems. However, our prediction strategy does not assume any particular event model. An operating system scheduler manages all these tasks and decides which task gets to run on the processor. The average rate at which events arrive at the processor is $\lambda = \sum \lambda_k$. The processor in turn offers a time varying processing rate, $\mu(r)$. The operating system kernel measures the idle cycles and computes the normalized workload, w, over some observation frame. The workload monitor sets the processing rate, r, based on the current workload, w, and a history of workloads from previous observation frames. This rate, r, in turn decides the operating frequency, $f(r)$, which in turn determines the operating voltage, $V(r)$, for the next observation slot.

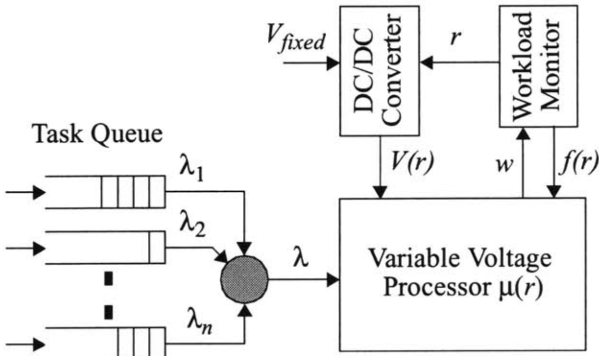


Figure 2-4: Block diagram of a DVS processor system

The problems that we address in this chapter are: (i) What kind of future workload prediction strategy should be used? (ii) What is the duration of the observation slot, i.e., how frequently should the processing rate be updated? The overall objective being to minimize energy consumption under a given performance requirement constraint.

2.2.2 Frequency and Minimum Operating Voltage

The gate delay of a simple CMOS inverter is given by

$$t_p \approx \frac{C_L}{2V_{dd}}\left(\frac{1}{k_p} + \frac{1}{k_n}\right) \tag{2-3}$$

where k_p and k_n are the gain factors of the PMOS and NMOS devices [9]. Therefore, gate delays in general scale inversely with the operating voltage. The worst case delay in a processor is simply the addition of similar delays terms from various circuit blocks in the crit-

ical path. This worst case delay determines the maximum operating frequency of the processor, $f \propto 1/t_{d,max}$. As such, the measured relation between minimum operating voltage and frequency is almost linear. The minimum measured operating voltage and corresponding frequency points have been plotted on a normalized scale for the StrongARM SA-1100 and the Pentium III processors in Figure 2-5. Most processor systems will have a discrete set of operating frequencies which implies that the processing rate levels are quantized. The StrongARM SA-1100 microprocessor, for instance, can run at 10 discrete frequencies in the range of 59 MHz to 206 MHz [16]. As we shall show later, discretization of the processing rate does not significantly degrade the energy savings from DVS.

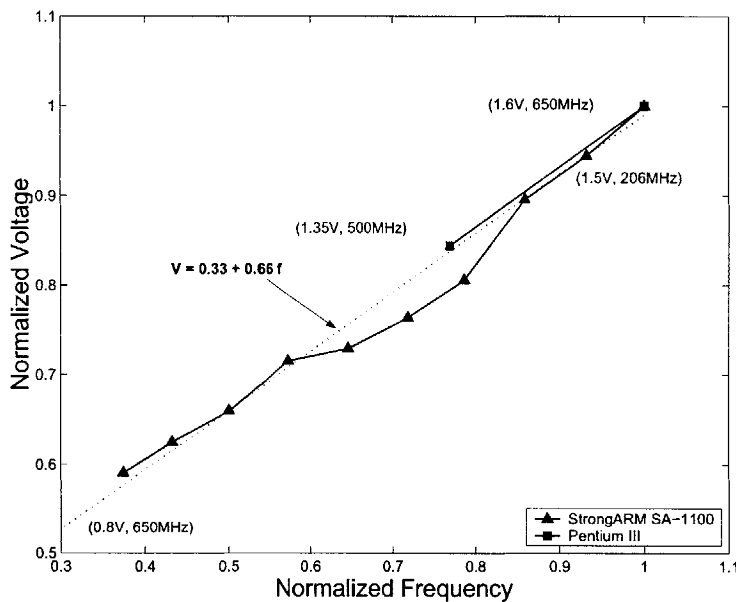


Figure 2-5: Frequency and minimum operating voltage (normalized plot)

2.2.3 Markov Processes

A stochastic process is called a Markov process if its past has no influence on its future once the present is specified [37]. Consider the sequence $X[k] = aX[k\text{-}1] + n[k]$, where $n[k]$ is a white noise process. Clearly, at instance k, the process $X[k]$, does not depend on any information prior to instance k-1. The precise definition of this limited form of historical dependency is as follows: $\boldsymbol{X}$ is an Nth order Markov process if its probability distribution function $P_{X[k]}$ satisfies

$$P_{X[k]}(y|X[k-1],X[k-2],...,X[0]) = P_{X[k]}(y|X[k-1],X[k-2],...,X[k-N]) \quad (2\text{-}4)$$

i.e., the most recent N values contain all the information about the past evolution of the process that is needed to determine the future distribution of the process.

Markov processes have been used in the context of Dynamic Power Management (DPM). In [38] a continuous-time, controllable Markov process model for a power managed system is introduced and DPM is formulated as a policy optimization problem. We propose to use Markov processes in the context of workload prediction, i.e., we propose to predict the workload for the next observation interval based on workload statistics of the previous N intervals.

2.2.4 Prediction Algorithm

Let the observation period be T. Let $w[n]$ denote the average normalized workload in the interval $(n-1)T \leq t < nT$. At time $t = nT$, we must decide what processing rate to set for the next slot, i.e., $r[n+1]$, based on the workload profile. Our workload prediction for the $(n+1)^{th}$ interval is given by

$$w_p[n+1] = \sum_{k=0}^{N-1} h_n[k]w[n-k] \quad (2\text{-}5)$$

where $h_n[k]$ is an N-tap, adaptable FIR filter whose coefficients are updated in every observation interval based on the error between the processing rate (which is set using the workload prediction) and the actual value of the workload.

Let us assume that there are L discrete processing levels available such that

$$r \in R_L, R_L = \left[\frac{1}{L}, \frac{2}{L}, ..., 1\right] \quad (2\text{-}6)$$

where we have assumed a uniform quantization interval, $\Delta = 1/L$. We have also assumed that the minimum processing rate is $1/L$ since $r = 0$ corresponds to the complete off state. Based on the workload prediction, $w_p[n+1]$, the processing rate, $r[n+1]$, is set such that

$$r[n+1] = \left\lceil \frac{w[n+1]}{\Delta} \right\rceil \Delta \quad (2\text{-}7)$$

i.e., the processing rate is set to a level just above the predicted workload.

2.2.5 Type of Filter

We explored a variety of possible filters for our prediction scheme and compared their performance. In this section we outline the basic motivation behind the top four filters and later present results showing the prediction performance of each of them.

- **Moving Average Workload** (MAW) - The simplest filter is a time-invariant moving average filter, $h_n[k] = 1/N$ for all n and k. This filter predicts the workload in the next slot as the average of the workload in the previous N slots. The basic motivation is that if the workload is truly an Nth order Markov process, averaging will result in workload noise being removed by low pass filtering. However, this scheme is too simplistic and may not work with time varying workload statistics. Also, averaging results in high-frequency workload changes being removed and, as a result, instantaneous performance hits are high.
- **Exponential Weighted Averaging** (EWA) - This filter is based on the idea that effect of the workload k-slots before the current slot lessens as k increases, i.e., it gives maximum weight to the previous slot, lesser weight to the one before, and so on. The filter coefficients are $h_n[k] = a^{-k}$, for all n, with a chosen such that $\sum h_n[k] = 1$ and a is positive. The idea of exponential weighted averaging has been used in the prediction of idle times for dynamic power management using shutdown techniques in event driven computation [39]. There too the idea is to assign progressively decreasing importance to historical data.
- **Least Mean Square** (LMS) - It makes more sense to have an adaptive filter whose coefficients are modified based on the prediction error. Two popular adaptive filtering algorithms are the Least-Mean-Square (LMS) and the Recursive-Least-Squares (RLS) algorithms [41]. The LMS adaptive filter is based on a stochastic gradient algorithm. Let the prediction error be $w_e[n] = w[n] - w_p[n]$, where $w_e[n]$ denotes the error and $w[n]$ denotes the actual workload as opposed to the predicted workload $w_p[n]$ from the previous slot. The filter coefficients are updated according to the following rule

$$h_{n+1}[k] = h_n[k] + \mu w_e[n] w[n-k] \qquad (2\text{-}8)$$

where μ is the step size. Use of adaptive filters has its advantages and disadvantages. On one hand, since they are self-designing, we do not have to worry about individual traces. The filters can 'learn' from the workload history. The obvious problems involve convergence and stability. Choosing the wrong number of coefficients or an inappropriate step size may have very undesirable consequences. RLS adaptive filters differ from LMS adaptive filters in that they do not employ gradient descent. Instead they employ a clever result from linear algebra. In practice they tend to converge much faster but they have higher computational complexity.

- **Expected Workload State** (EWS) - The last technique is based on a pure probabilistic formulation and does not involve any filtering. Let the workload be discrete and quantized like the processing rate as shown in Equation 2-6 with the state 0 also included. The error can be made arbitrarily small by increasing the number of levels, L. Let $\mathbf{P} = [p_{ij}]$, $0 \le i \le L$, $0 \le j \le L$,denote a square matrix with elements p_{ij} such that $p_{ij} = Prob\{\ w[r+1] = w_j \mid w[r] = w_i\ \}$ where w_k represents the kth workload level out of the L+1 discrete levels. Therefore $\mathbf{P}$ is the state transition matrix with the property that $\sum_j p_{ij} = 1$. The workload is then predicted as

$$w_p[n+1] = \mathbf{E}\{w[n+1]\} = \sum_{j=0}^{L} w_j p_{ij} \tag{2-9}$$

where $w[n] = w_i$ and $\mathbf{E}\{.\}$ denotes the expected value. The probability matrix is updated in every slot by incorporating the actual state transition. In general the $(r+1)^{th}$ state can depend on the previous N states (as in an Nth order Markov process) and the probabilistic formulation is more elaborate.

Figure 2-6 shows the prediction performance using Root-Mean-Square error as an evaluation metric for the four different schemes. If the number of taps is small the prediction is too noisy and if it is too large there is excessive low pass filtering. Both result in poor prediction. In general we found that the LMS adaptive filter outperforms the other techniques and produces best results with $N = 3$ taps. The adaptive prediction of the filter is shown for a workload snapshot in Figure 2-7.

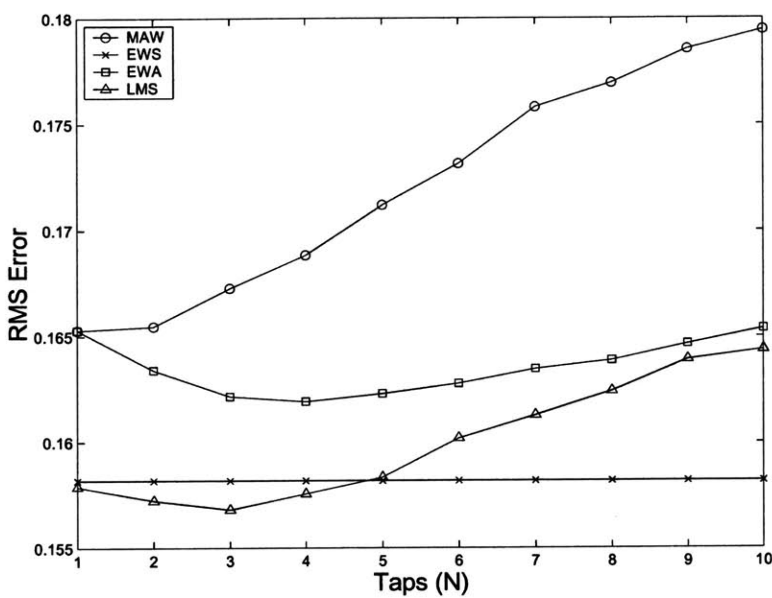


Figure 2-6: Prediction performance of the different filters

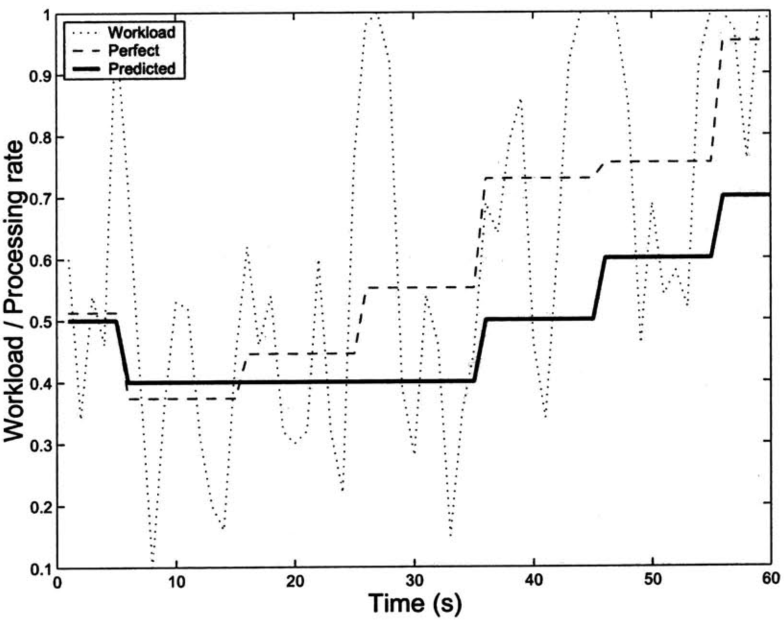


Figure 2-7: Workload tracking by the LMS filter

2.3 Energy Performance Trade-offs

2.3.1 Performance Hit Function

Definition: The performance hit, $\phi(\Delta t)$, over a time frame Δt, is defined as the extra time (expressed as a fraction of Δt) required to process the workload over time Δt at the processing rate available in that time frame.

Let $\bar{w}_{\Delta t}$ and $\bar{r}_{\Delta t}$ respectively denote the average workload and processing rates over the time frame of interest, Δt. The extra number of cycles required, assuming $\bar{w}_{\Delta t} > \bar{r}_{\Delta t}$, to process the entire workload is ($\bar{w}_{\Delta t} f_{max} \Delta t$ - $\bar{r}_{\Delta t} f_{max} \Delta t$) where f_{max} is the maximum operating frequency. Therefore the extra amount of time required is simply ($\bar{w}_{\Delta t} f_{max} \Delta t$ - $\bar{r}_{\Delta t} f_{max} \Delta t$) / $\bar{r}_{\Delta t} f_{max}$. Therefore,

$$\phi(\Delta t) = \frac{(\bar{w}_{\Delta t} - \bar{r}_{\Delta t})}{\bar{r}_{\Delta t}} \tag{2-10}$$

If $\bar{w}_{\Delta t} < \bar{r}_{\Delta t}$ then the performance penalty is negative. The way to interpret this is that it is a slack or idle time. Using this basic definition of performance penalty we define two different metrics: $\phi^{T}_{max}(\Delta t)$ and $\phi^{T}_{avg}(\Delta t)$ which are respectively the maximum and average performance hits measured over Δt time slots spread over an observation period T as shown in Figure 2-8.

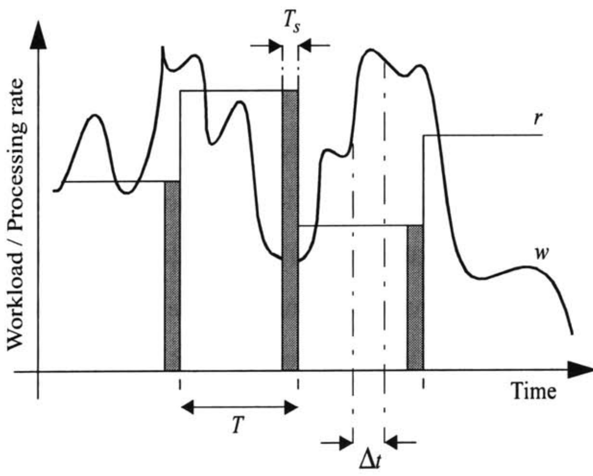


Figure 2-8: Performance hit, settling time notions

Figure 2-9 shows the average and maximum performance hit as a function of the update time T, for prediction using $N = 2$, 6 and 10 taps. The time slots used were $\Delta t = 1$s

and the workload trace was that of the dialup server. The results have been averaged over 1 hour. While the maximum performance hit increases as T increases, the average performance hit decreases. This is because as T increases the excess cycles from one time slot spill over to the next one and if the slot has a negative performance penalty (i.e., slack / idle cycles) then the average performance hit over the two slots decreases and so on. On the other hand, as T increases, the chances of an increased disparity between the workload and processing rate in a time slot is more and the maximum performance hit increases.

This leads to a fundamental energy-performance trade-off in DVS. Because of the convexity of the $E(r)$ relationship and Jensen's inequality, we would always like to work at the overall average workload. Therefore, over a 1 hour period for example, the most energy efficient DVS solution is one where we set the processing rate equal to the overall average workload over the 1 hour period. In other words, increasing T leads to increased energy efficiency (assuming perfect prediction). On the other hand, increasing T, also increases the maximum performance hit. In other words, the system might be sluggish in moments of high workload. Maximum energy savings for a given performance hit involves choosing the maximum update time, T, such that the maximum performance hit is within bounds as shown in Figure 2-9[1].

In most DVS processors, there is a latency overhead involved in processing rate update. This is because there is a finite feedback bandwidth associated with the DC/DC converter. Normally a good voltage regulator can switch between voltage output levels in a few tens of microseconds [35]. Changing the processor clock frequency also involves a latency overhead during which the PLL circuits lock. In general, to be on the safe side, voltage and clock frequency changes should not be done in parallel. While switching to a lower processing rate, the frequency should first be decreased and subsequently the voltage should be lowered to the appropriate value. On the contrary, switching to a higher processing rate requires the voltage to be increased first followed by the frequency update. This ensures that the voltage supply to the processor is never lower than the minimum required for the current operating frequency and avoids data corruption due to circuit fail-

1. Although we have used the maximum performance hit function for choosing the optimum update time T this might be very pessimistic. It may be a more energy efficient to relax the update time T such that even if we do not meet the worst case performance requirement, we are still able to do it in most cases.

ure. However, in [42] the update is done in parallel because the converter and the clock update latency are comparable (approximately 100μs) and it still works.

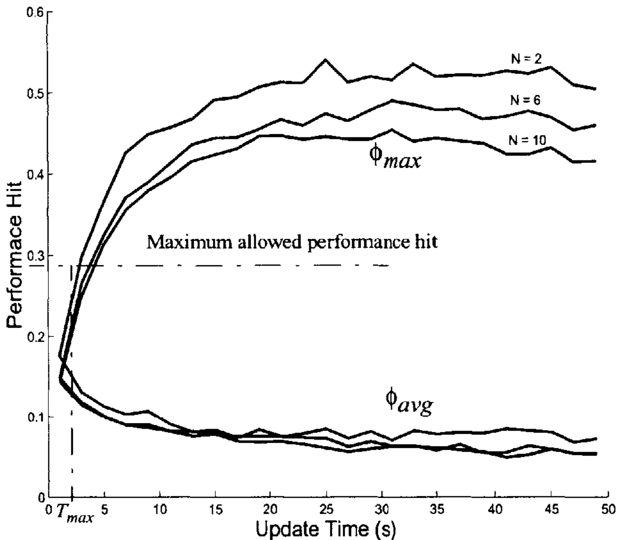


Figure 2-9: Average and maximum performance hits

We denote the processing rate update latency by T_s (for settling time). It is possible to incorporate this overhead in the performance hit function. Over the update time T, the extra number of cycles is now equal to ($\bar{w}_{\Delta t} f_{max} T$ - $\bar{r}_{\Delta t} f_{max}(T - T_s)$) and the corresponding performance hit function becomes

$$\phi(T) = \frac{\left(\bar{w}_{\Delta t} - \bar{r}_{\Delta t}\left(1 - \frac{T_s}{T}\right)\right)}{\bar{r}_{\Delta t}} \tag{2-11}$$

In our experiments, the time resolution for workload measurement was 1 second. Since we want to work at averaged workload this is not a problem unless there are very stringent real-time requirements. The other advantage of using a lower time resolution is that the workload measurement subroutine does not itself add substantial overhead to the workload if the measurement duty-cycle is small. The update latency is of the order of 100 μs and since this is insignificant compared to our minimum update time we have used Equation 2-10 instead of Equation 2-11.

2.3.2 Optimizing Update Time and Taps

The conclusion that increasing the update time, T, results in the most energy savings is not completely true. This would be the case with a perfect prediction strategy. In reality if the update time is large, the cost of an overestimated rate is more substantial and the energy savings decrease. Since we are using discrete processing rates (in all our simulations the number of processing rate levels is set to 10 unless otherwise stated), and we round off the rate to the next higher quanta, using a larger update time results in higher overestimate cost. A similar argument holds for the number of taps, N. A very small N implies that the workload prediction is very noisy and the energy cost is high because of widely fluctuating processing rates. A very large N on the other hand implies that the prediction is heavily low-pass filtered and therefore sluggish to rapid workload changes. This leads to higher performance penalty. Figure 2-10 shows the relative energy plot (normalized to the no DVS case) for the dialup server trace. The period of observation was 1 hour. The energy savings showed a 13% variation based on what N and T were chosen for the adaptive filter. The implications of the above discussion is at once apparent.

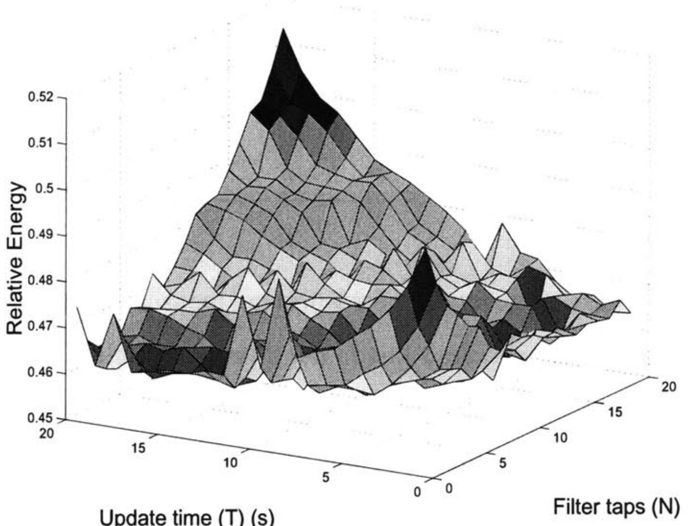


Figure 2-10: Energy consumption (normalized to the no DVS case) as a function of update time and prediction filter taps

2.4 Results

Table 2-1: DVS energy savings ratio ($E_{no\text{-}dvs}/E_{dvs}$) [N = 3, T = 5 s]

Trace	Filter	ESR			ϕ_{avg} (%)	ϕ_{max} (%)
		Perfect	Actual	Perfect / Actual		
Dialup Server	MAW	1.4	1.3	1.1	10.6	34.8
	EWS		1.2	1.1	10.8	36.3
	EWA		1.3	1.1	10.6	35.4
	LMS		1.4	1.0	14.7	43.1
File Server	MAW	2.7	1.9	1.4	12.6	42.8
	EWS		1.8	1.5	7.4	33.8
	EWA		1.9	1.4	9.2	37.4
	LMS		2.2	1.2	14.1	47.7
User Work Station	MAW	3.6	2.5	1.4	3.6	35.3
	EWS		2.8	1.3	3.8	35.1
	EWA		2.5	1.5	3.7	35.6
	LMS		2.5	1.4	3.9	36.0

Table 2-1 summarizes our key results. We used 1 hour workload traces from three different types of machines over different times of the day. Their typical workload profiles are shown in Figure 2-2. The Energy Savings Ratio (ESR) is defined as the ratio of the energy consumption with no DVS (simple frequency scaling) to the energy consumption with DVS. Maximum savings occur when we set the processing rate equal to the average workload over the entire period. Maximum savings is not usually achievable because of two reasons: (i) The maximum performance hit increases as the averaging duration is increased, and (ii) It is impossible to know the average workload over the stipulated period *a priori*. The filters have $N = 3$ taps and an update time $T = 5$ s, based on our previous discussion and experiments performed. The 'Perfect' column shows the ESR for the case where we had a perfect predictor for the next observation slot. The 'Actual' column shows the ESR obtained by the various filters. In almost all our experiments the LMS filter gave

the best energy savings. The last two columns are the average and maximum performance hits. The average performance hit is around 10% while the maximum performance hit is about 40%.

Finally, the effect of processing level quantization is shown in Figure 2-11. As the number of discrete levels, L, is increased, the ESR gets closer to the perfect prediction case. For $L = 10$ (as available in the StrongARM SA-1100) the ESR degradation due to quantization noise is less than 10%

.

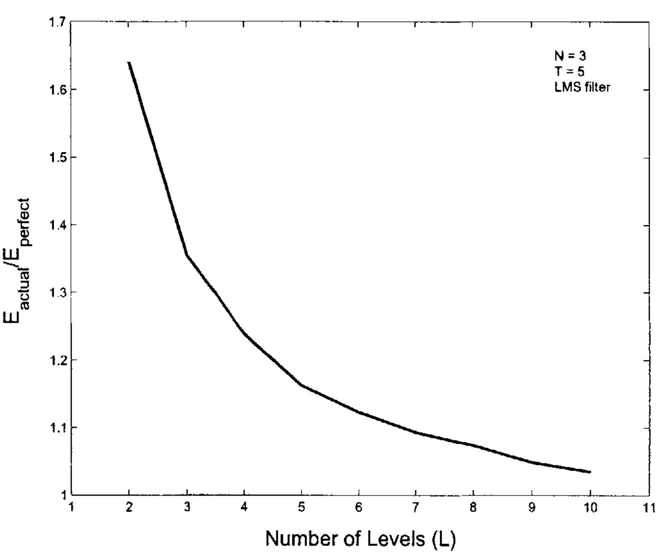


Figure 2-11: Effect of number of discrete processing levels, L

2.5 Summary of Contributions

Dynamic voltage and frequency scaling an effective technique to reduce processor energy consumption without causing significant performance degradation. In the coming years, most energy conscious processors will allow dynamic voltage and frequency change at runtime. We demonstrated, using trace data from three different processors running different kinds of tasks, that energy savings by a factor of two to three is possible on low workload processors (compared to the case where only frequency is adapted). We also showed that maximum energy savings occur if the processing rate is set to the overall average workload. This, however, is generally infeasible *a priori* and even if possible leads to high performance penalties. Frequent processing rate updates ensure that the per-

formance penalty is limited. The faster the update rate, the lower the energy savings and the lesser the performance penalty. Workload prediction is required to set the processing rate for the next update slot. We developed an adaptive filtering based workload prediction scheme that is able to track workload changes and speculate future variations. Such strategies will have to be incorporated into the dynamic voltage and frequency setting module of the operating system. The loss in energy savings due to quantization of the available operating frequencies in the processor was analyzed and it has been shown that the inefficiency introduced is quite nominal.

Chapter 3

Power Management in Real-Time Systems

Real-time systems are defined as systems where both computational correctness and time of completion are critical. A simple real-time system might be a video decoder where 30 frames must be decoded every second for uninterrupted viewing. Real-time systems are of two types - hard real-time and soft real-time systems. A hard real-time system is one where catastrophic failure can result if a computation is not completed before its deadline. A soft real-time system, on the other hand, will have degradation in quality of service if deadlines are not met. In the last chapter, operating system directed power management was discussed for non real-time systems. It was shown that dynamic voltage and frequency control along with a workload prediction scheme can be effective employed to reduce energy consumption with little visible performance loss. The proposed technique is good for systems where no real-time constraints exist since it does not consider any deadlines that particular tasks might have.

The job of a Real-Time Operating System (RTOS) is to schedule tasks to ensure that all tasks meet their respective deadlines. Real-time scheduling can be broadly classified into *static* and *dynamic* algorithms. Static algorithms are applicable to task sets where complete information (e.g., arrival times, computation time, deadlines, precedence, dependencies, etc.) is available *a priori*. The Rate Monotonic (RM) algorithm one such algorithm and is optimal among all fixed priority assignments in the sense that no other fixed priority algorithm can schedule a task set that cannot be scheduled by RM [43]. Dynamic scheduling is characterized by inherent uncertainty and lack of knowledge about the task set and its timing constraints. The Earliest Deadline First (EDF) algorithm has been shown to be an optimal dynamic scheduling algorithm [44]. However, EDF assumes resource sufficiency (i.e., even though tasks arrive unpredictably, the system resources have a sufficient *a priori* guarantee such that at any given time all tasks are schedulable) and in the absence of such a guarantee the EDF performance degrades rapidly in the presence of

overload. The Spring algorithm has been proposed for such dynamic resource insufficient environments and uses techniques such as admission control and planning-based algorithms [45].

In [46] optimal *off-line* scheduling techniques for variable voltage/frequency processors is analyzed for independent tasks with arbitrary arrivals. The authors of [47] have proposed a set of heuristic algorithms to schedule a mixed workload of periodic and sporadic tasks. In this chapter, we discuss energy efficient real-time scheduling algorithms that can exploit the variable voltage and frequency hooks available on processors for improving energy efficiency and therefore battery life of embedded systems. We propose the *Slacked Earliest Deadline First (SEDF)* algorithm and prove that it is optimal in minimizing processor energy consumption *and* maximum lateness for an independent arbitrary task set [49]. We also derive an upper bound on energy savings through dynamic voltage and frequency scaling for all possible algorithms and arrival statistics. The SEDF algorithm is dynamic and approaches the EDF algorithm as processor utilization increases. We use the EDF algorithm as a baseline to compare the scheduling performance of SEDF. Optimal processor voltage and frequency assignments for periodic tasks is also discussed with the EDF and RM algorithms used as a baseline.

3.1 Aperiodic Task Scheduling

3.1.1 Performance Evaluation Metrics in Real-Time Systems

The performance of a real-time scheduling algorithm is evaluated with respect to a cost function defined over the task set. A typical task set consists of N tasks where the ith task is characterized by an arrival time, a_i, a computation time, c_i, and a deadline, d_i. The time of completion of the task is denoted by f_i. The metric adopted in a scheduling algorithm can have strong implications on the performance of a real-time system and must be carefully chosen according to the specific requirements of the application [48]. Table 3-1 lists some common cost function metrics. The average response time is not generally of interest in real-time systems since it does not account for deadlines. The same is true for total completion time. The weighted sum of completion times is relevant when tasks have different priorities and the effect of completion of a particular task is attributed a measure of significance. Minimizing maximum lateness can be useful at design time when

resources can be added until the maximum lateness achieved on a task set is less than or equal to zero. In that case, no task misses its deadline. In general, however, minimizing maximum lateness does not minimize the number of tasks that miss their deadlines. In soft real-time systems, it is usually better to minimize the number of late tasks. For example, in video decoding the visual quality depends on the number of frames that do not get decoded within the deadline more significantly than the maximum time by which a frame decoding misses its deadline. If a deadline is missed, it is better to throw the frame out anyway. On the other hand, maximum lateness, L_{max}, is a good criteria for hard real-time algorithms as it upper bounds the time by which any task misses its deadline. It is a worst case performance metric. We will use L_{max} as a metric to evaluate our scheduling algorithm.

Table 3-1: Real-time performance metrics

Metric	Cost Function
Average response time	$\bar{t}_r = \frac{1}{n}\sum_{i=1}^{N}(f_i - a_i)$
Total completion time	$t_c = \max_i(f_i) - \min_i(a_i)$
Weighted sum of completion times	$t_w = \sum_{i=1}^{N} w_i f_i$
Maximum lateness	$L_{max} = \max_i(f_i - d_i)$
Maximum number of late tasks	$_{late} = \sum_{i=1}^{N}\text{miss}(f_i)$ $\quad \text{miss}(f_i) = \begin{cases} 0, f_i \le d_i \\ 1, \text{otherwise} \end{cases}$

3.1.2 The Earliest Deadline First Algorithm

Let the task set to be scheduled be denoted by

$$\vartheta = \{\tau_i(a_i, c_i, d_i), 0 < i \le N\} \tag{3-1}$$

We assume that the tasks are independent, i.e., they do not have any dependence constraints, the system consists on one processor and preemption is allowed. Under these conditions, the following theorem holds.

Theorem I: *Given a set of N independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness* [44].

This theorem, known as the *Earliest Deadline First* (EDF) algorithm, was first posed by Horn [44], and was proved to be optimal by Dertouzos [50].

3.1.3 Real-Time Systems with Variable Processing Rate

With variable voltage/frequency systems two things need to be determined at every scheduling interval - (i) The task to be scheduled, and (ii) The relative processing rate. A simple greedy algorithm that sets the processing rate such that the scheduled task just meets its deadline will not work. This can be illustrated by a simple example shown in Figure 3-1.

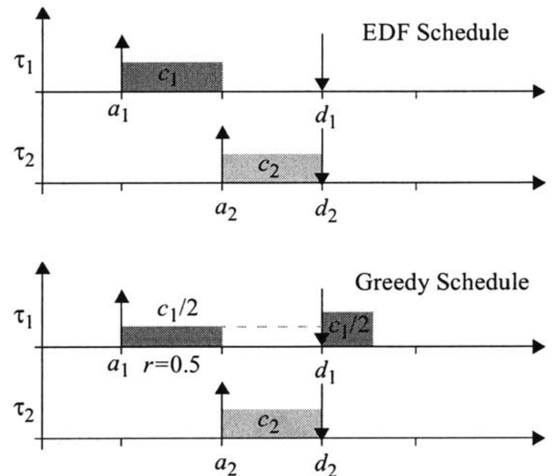


Figure 3-1: Greedy scheduling of processing rate

The two tasks have the same deadline and one comes in after the other one. EDF is able to schedule the two tasks such that both of them meet their respective deadlines. A greedy algorithm that sets the processing rate, r, based on information about the current task's deadline is not able to schedule the tasks. At time $t = a_1$, the greedy scheduler sees only task τ_1 with deadline d_1 and sets the processing rate to $r = c_1/(d_1\text{-}a_1)$, such that the task occupies the complete time available. At time $t = a_2 < d_1$, τ_2 arrives with the same

deadline $d_1 = d_2$, and even though the rate is set back to $r = 1$ and τ_2 meets its deadline, τ_1 fails to complete before its deadline d_1. Therefore any algorithm that modifies processing rate must do so in an intelligent way.

3.1.4 The Slacked Earliest Deadline First (SEDF) Algorithm

In this section we propose the SEDF algorithm and show that it is optimal is minimizing processor energy and maximum lateness. In fact, the SEDF algorithm approaches the EDF algorithm in an asymptotic way as the processor is fully utilized.

Theorem II: *Given a set of independent tasks with arbitrary arrival times, computation times and deadlines, any algorithm that at every scheduling instant t_i executes the task with the earliest absolute deadline among all the ready tasks and sets the instantaneous processing rate to $r_i(S_i, U_i)$, where U_i is the processor utilization up to time t_i and S_i is the available slack for the scheduled task, is optimal with respect to minimizing the maximum lateness and processor energy. This optimum processing rate is approximated by*

$$r_i(S_i, U_i) = \begin{cases} S_i + (1 - S_i)U_i, 0 < S_i \le 1 \\ 1, \text{otherwise} \end{cases} \quad (3\text{-}2)$$

Proof: Let the scheduling intervals be Δt, such that a decision as to which task will be allocated to the processor during the interval (t_i, t_{i+1}) is made at discrete time instant $t_i = i\Delta t$. The problem we want to solve is: Which task should be allocated to the processor during the interval (t_i, t_{i+1}) and what should the relative processing rate, $0 \le r_i \le 1$, be? We assume that the scheduling decision takes negligible time compared to the scheduling interval Δt and that the computation times are integral multiples of the scheduling interval.

Let τ_i be the task with the earliest deadline at time instant t_i and let $\bar{c}_i$ denote the residual computation time ($\bar{c}_i = c_i$ if the task τ_i was never scheduled before). The computation time is always with reference to the maximum processing rate, i.e., $r = 1$. Let U_i denote the processor utilization up to time t_i. U_i is simply the ratio of the number of idle frames (where no tasks were ready for scheduling) to the total number of scheduling frames, i. Let $\bar{d}_i = d_i - t_i$, be the maximum number of scheduling slots available to task τ_i to complete before its deadline. If $\bar{d}_i < 0$, the task has missed its deadline and there is no positive slack

available. The available slack for task τ_i is the ratio of $\bar{c}_i$ to $\bar{d}_i$ and is denoted by S_i. All these variables are shown in Figure 3-2. If $S_i > 1$, the task will miss its deadline no matter what. If $S_i < 0$, the task has already missed its deadline. Under both these circumstances, minimizing maximum lateness requires that the task be finished as soon as possible and so the processing rate r_i is set to 1. Note that $S_i = 0$ is not possible since that would mean that the task has zero residual computation time, i.e., it is already completed.

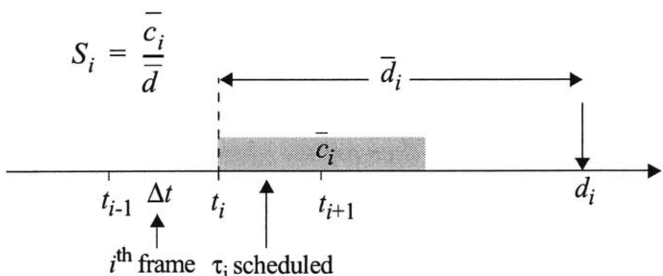


Figure 3-2: Illustrating the parameters involved in SEDF

For the case where $0 < S_i \le 1$, the analysis is as follows. Assuming that the processor utilization is stationary over the next $\bar{d}_i$ slots, the probability that the task will finish before its deadline at the maximum processing rate is given by

$$\text{Prob}[\tau_i \text{ finishes}] = P(r = 1) = \sum_{k = \bar{c}_i}^{d_i} {}^{\bar{d}_i}C_k (1 - U_i)^k U_i^{\bar{d}_i - k} \tag{3-3}$$

which follows from the fact at the maximum processing rate there are $\bar{c}_i$ slots required to complete the task out of a maximum of $\bar{d}_i$ slots and the probability of any particular slot being occupied is U_i. The probability of completion (before the deadline), at any processing rate, r, is therefore given by

$$P(r) = \sum_{k = \left\lceil \frac{\bar{c}_i}{r} \right\rceil}^{d_i} {}^{\bar{d}_i}C_k (1 - U_i)^k U_i^{\bar{d}_i - k} \tag{3-4}$$

where the number of required slots simply scales with the reduced processing rate. The energy savings at any processing rate, r, for a given task is given by

$$E_{save}(r) = 1 - r^2 \tag{3-5}$$

based on a simplified version of the energy workload model proposed in Section 2.1. Let us define

$$\xi(r) = P(r) \cdot E_{save}(r) \tag{3-6}$$

$\xi(r)$ can be interpreted as the expected energy savings given the task completed before its deadline. To maximize $\xi(r)$ we set the partial derivative with respect to r equal to zero,

$$\frac{\partial}{\partial r}\xi(r) = 0 \quad \Rightarrow \quad \frac{2r}{1-r^2} = \frac{P'(r)}{P(r)} \tag{3-7}$$

The optimum r cannot be obtained analytically since $P(r)$ is not differentiable in the entire range $0 \le r \le 1$. Figure 3-3 shows the completion probability, the weighted energy savings as a function of the processing rate, r, and the optimum processing rate as a function of the processor utilization (for a slack $S_i = 0.1$). As r increases, the computation slots required decreases and the probability, $P(r)$, of completion *increases*. The increase is faster with lower processor utilization. The energy savings, on the other hand, *decreases* with increased processing rate. The weighted energy savings therefore has an optimum processing rate where it is maximized. Figure 3-4 shows the optimized processing rate, r, as a function of the processor utilization, U, and the available slack, S. This is an exact numerical solution for Equation 3-7. Also shown in Figure 3-4 is the optimum processing rate as a linear function of U and S as represented by Equation 3-2. A closed form expression for optimum r can be obtained if we let $\Delta t \to 0$ and in the limit, the function $P(r)$ becomes continuous. Using Stirling's approximation,

$$n! \approx \sqrt{2\pi n}\, n^n e^{-n} \tag{3-8}$$

in Equation 3-4, the limit of the sum becomes an integral and for U around 0.5, the probability of completion is given by the Gaussian integral (the error function).

$$P(r) = \frac{1}{\sqrt{2\pi}}\int_a^1 e^{-\frac{x^2}{2}} dx \qquad a = \frac{\frac{S_i}{r} - (1-U)\bar{d}_i}{\sqrt{\bar{d}_i(U(1-U))}} \tag{3-9}$$

for values of U close to 0, the function tends to a Poisson integral. Although these equations can be solved exactly, the simple linear function shown in Equation 3-2 is quite adequate as we have shown in Figure 3-4 and will show in our results.

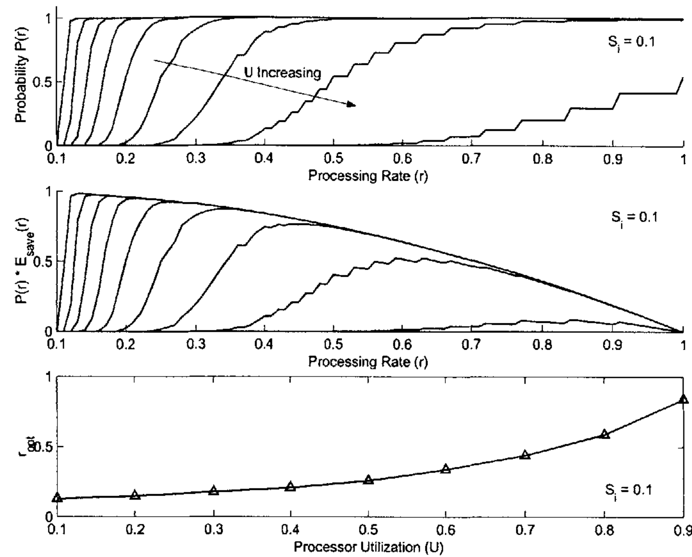


Figure 3-3: Completion probability, weighted energy and optimum processing rate

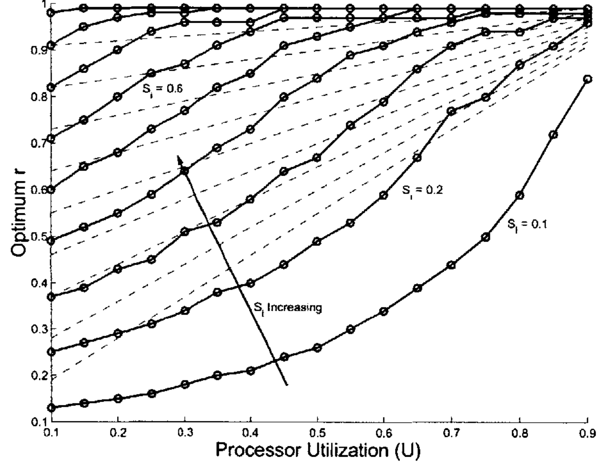
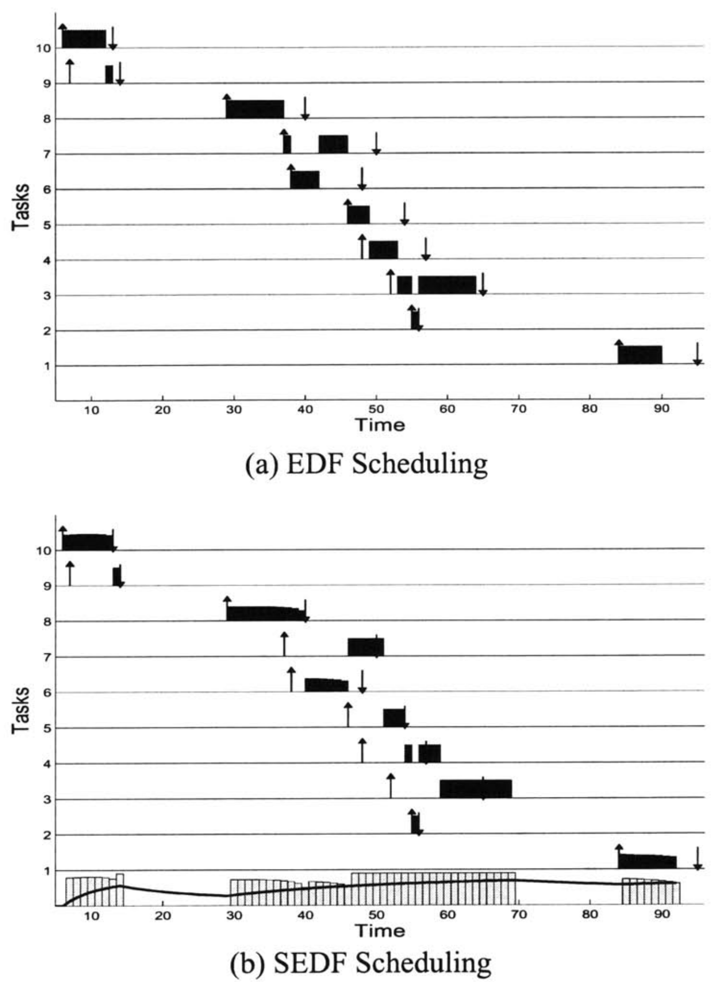


Figure 3-4: Optimum processing rate as a function of processor utilization and slack

3.1.5 Results

Figure 3-5 shows a simulated example of EDF and SEDF scheduling on a set of 10 tasks characterized by a uniform random process. While EDF meets all deadlines (the preemptive nature is obvious from tasks 3 and 7) SEDF is not able to meet all deadlines, the L_{max} being equal to 3 time units. The energy savings is 53%. The changing height of the computation time bars indicates reduced processing rate which is shown along with the

evolving processor utilization at the bottom of the graph in Figure 3-5(b). Since SEDF is stochastically optimal, the maximal lateness and energy savings improve as the learning time and task set increases.



(a) EDF Scheduling

(b) SEDF Scheduling

Figure 3-5: Example of EDF and SEDF scheduling

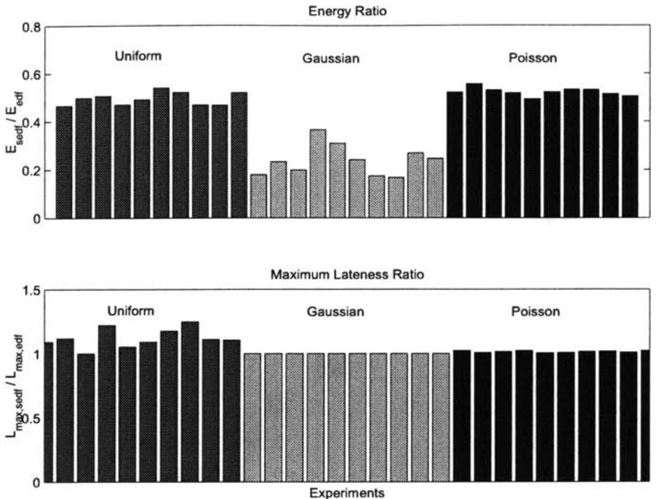


Figure 3-6: Comparing the performance of EDF and SEDF

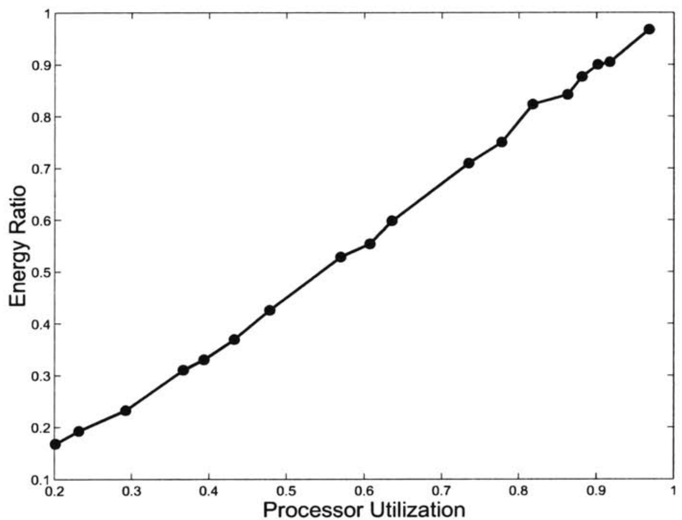


Figure 3-7: Energy ratio as a function of processor utilization

We have compared the SEDF algorithm to the EDF algorithm based on random task sets where the arrival times, computation times and deadlines are characterized by uniform, Gaussian and Poisson processes. In each case, the maximum lateness and the energy consumption were compared. The energy savings averaged over 3×10^6 experiments was about 60% while the degradation in maximal lateness was less that 10%. The results of all the experiments have been summarized in Figure 3-6 where each bar represents the average of 10^5 experiments. The increased energy savings from Gaussian characterized task

sets can be attributed to the fact that arrivals and computations are more clustered (i.e., within $\pm 2\sigma$ mostly) and so the predicted slack is better. Finally, Figure 3-7 shows the ratio of the energy consumption of the SEDF to the EDF case as a function of processor utilization. As the utilization increases, slacking is reduced and the SEDF schedule tends to the EDF schedule with processing rate increasingly being set to 1.

3.1.6 Upper Bound on Energy Savings

Theorem III: *Given a set of independent tasks with arbitrary arrival times, computation times and deadlines, the maximum energy savings possible using any dynamic voltage and frequency setting algorithm which produces a schedule that meets all deadlines is bounded by $E_{save}(r_{min})$, where the processing rate*

$$r_{min} = \frac{\sum c_i}{\max(d_i)} \tag{3-10}$$

Proof: The denominator term of r_{min} is simply the maximum possible total time, T, allowed to finish all the tasks before their respective deadlines, i.e., $r_{min} = (\sum c_i)/T$. It is obvious that minimum energy results when tasks are slacked such that the entire time frame T is used up (i.e., processor utilization is 1). Assume that there exists an algorithm Λ, which is able to meet all deadlines and it schedules processing rates and tasks in each scheduling interval. A particular task τ_k, might get scheduled in different slots with different processing rates. Let the average processing rate seen by τ_k be $\bar{r}_k$. The actual computation time of τ_k is therefore $c_k/\bar{r}_k$, and therefore the absolute best case occurs when $T = \sum c_k/\bar{r}_k$. We will now show that minimum energy consumption occurs when all the processing rates are equal. We begin with the following inequality (which can be readily verified using the Cauchy-Schwarz inequality)

$$\left(\sum_k \frac{c_k}{\bar{r}_k}\right)\left(\sum_k c_k \bar{r}_k\right) \geq \left(\sum_k c_k\right)^2 \tag{3-11}$$

Rearranging the terms we get

$$\sum_k c_k \bar{r}_k \geq \left(\sum_k c_k\right)\left(\frac{\sum_k c_k}{T}\right) = \sum_k c_k r_{min} \qquad (3\text{-}12)$$

The total normalized energy consumption is

$$E_{tot} = \sum_k \frac{c_k}{\bar{r}_k} E(\bar{r}_k) = \sum_k c_k \bar{r}_k \qquad (3\text{-}13)$$

where we have substituted the quadratic energy consumption model of Figure 2-3. The left-hand side of Equation 3-12 is the energy consumption for the schedule produced by Λ, while the right-hand side is the energy consumption of a schedule where all tasks have the same processing rate, r_{min}. Therefore, it can be concluded that minimum energy consumption (or maximum energy savings) occurs when all tasks have the same averaged processing rate. Using a similar argument, it can be shown that within a task, minimum energy consumption occurs when each of the different scheduled processing rates are equal to the average processing rate, $\bar{r}_k$.

The maximum savings, for example, with the task set shown in Figure 3-5 is 74.5% (with $r_{min} = 0.5$). The savings by the SEDF algorithm was 53%. However, the comparison is not completely fair since the SEDF algorithm did not meet all the deadlines.

3.2 Periodic Task Scheduling

In this case our task set to be scheduled is denoted by

$$\vartheta = \{\tau_i(\phi_i, T_i, c_i), 0 < i \leq N\} \qquad (3\text{-}14)$$

where ϕ_i is the phase, T_i is the time period and c_i is the computation time of the ith task from a set of N periodic tasks. We assume that the tasks are independent, the system consists on one processor and preemption is allowed. Every task has to be executed once in each of its periods with the relative deadline being equal to the time period.

3.2.1 Dynamic Priority Assignments

Once again EDF is an optimal dynamic scheduling policy, reason being that EDF did not make any assumptions about tasks being periodic or aperiodic. EDF being intrinsically

preemptive, the currently executing task is preempted whenever another periodic instance with an earlier relative deadline becomes active. Since the task set, ϑ, is completely determined *a priori*, the processing rate can also be determined completely and does not have to be adaptive. In fact the following theorem holds.

Theorem IV: *A set of periodic tasks is guaranteed to be schedulable with maximum energy savings iff the processing rate is*

$$r_{min} = \sum_i \frac{c_i}{T_i} \tag{3-15}$$

Proof: It has been shown in [43] that a periodic task set is guaranteed to be schedulable by EDF *iff* $\sum_i c_i/T_i \leq 1$. Let $T = T_1 T_2 ... T_N$, be an observation time frame. Using a line of reasoning exactly similar to the proof of Theorem III, it can be shown that minimum processor energy consumption occurs when all tasks are slacked by the same amount, to the maximum allowable limit such that

$$\sum_i \frac{(c_i/r)}{T_i} \leq 1 \Rightarrow r_{min} = \sum_i \frac{c_i}{T_i} \tag{3-16}$$

3.2.2 Static Priority Assignments

It has been also shown in [43] that the Rate-Monotonic algorithm is an optimal fixed priority algorithm. RM schedules tasks based on their periods, with priorities statically assigned to be inversely proportional to the task periods (i.e., the highest priority being assigned to task having the smallest period and so on). Since priorities are statically assigned, a ready task with a lower period will preempt another task with a higher period despite the fact that its relative deadline is earlier. With such a fixed priority assignment, the following theorem holds.

Theorem V: *A set of N periodic tasks is guaranteed to be schedulable using fixed priority assignments with maximum energy savings if the processing rate is*

$$r_{min} = \frac{\sum_i \frac{c_i}{T_i}}{N(2^{1/N} - 1)} \tag{3-17}$$

Proof: It has been shown in [43] that RM guarantees that an arbitrary set of N periodic tasks is schedulable if the total processor utilization, $U = \sum_i c_i / T_i \leq N(2^{1/N} - 1)$. The processing rate in Equation 3-17 can be derived exactly as shown in the proof of Theorem IV.

3.3 Summary of Contributions

We analyzed energy efficient scheduling algorithms for arbitrary independent periodic and aperiodic task sets characterized by real-time deadlines using variable voltage and frequency assignments on a single processor. The Slacked Earliest Deadline First (SEDF) algorithm is proposed, and it is shown that SEDF is optimal in minimizing maximum lateness and processor energy consumption. A bound on the maximum energy savings possible with any algorithm, for a given task set, is also derived. Energy efficient scheduling for periodic task sets is also considered (both static and dynamic priority assignments) and optimal processing rate assignments were derived under a guaranteed schedulability criteria.

Chapter 4

Idle Power Management

A portable system spends a significant amount of time in a standby mode. For example, a cellular phone typically spends over 95% in the idle state (waiting for a call). A wireless sensor node can spend a lot of time waiting for a significant event to happen. Within a given system itself, different resources and blocks might be waiting for interrupts and service requests from other blocks. A common example is a hard disk drive waiting for read/write requests from the corresponding driver. From an energy savings perspective, it makes sense to shutdown a resource that is not being used. However, once the resource is shutdown, significant time and energy overheads might be required to wake it up and start using it again. If the overheads associated with turning a resource off/on were negligible, a simple greedy algorithm that shuts off the resource as soon as it is not required would be optimal. However, switching a resource off/on incurs an overhead and smarter algorithms that observe the usage profile of a resource to make shutdown/wakeup decisions are needed.

4.1 Previous Work

Researchers have tried to model the interarrival process of events in reactive systems. In [57] the distribution of idle and busy periods is represented by a time series and approximated by a least square regression model. In [39] the idleness prediction is based on a weighted sum of past periods where the weights decay geometrically. In [40] power optimization in several common hard real-time disk-based design systems is proposed. The authors of [58] use a stochastic optimization technique based on the theory of Markov processes to solve for an optimum power management policy.

While previous work has concentrated on prediction strategies for idle times, the granularity and overheads associated with shutdown has not been addressed. In this chapter, we propose and analyze a fine-grained shutdown scheme in the context of a sensor node [51]. The technique outlined is fairly general and can be used with little or no modification

in other systems characterized by event driven computation. We introduce the idea of a "power-aware" system description that describes various stages of shutdown in a device and captures the corresponding power and latency overheads associated with those shutdown modes. This models the system as a set of finite, power differentiated, multiple shutdown states, rather than just one on/off state.

4.2 Multiple Shutdown States

It is not uncommon for a device to have multiple power modes. For example, the StrongARM SA-1100 processor has three power modes - 'run', 'idle' and 'sleep' [52]. Each of these modes is associated with a progressively lower level of power consumption. The 'run' mode is the normal operating mode of the processor, all power supplies are enabled, all clocks are running and every on-chip resource is functional. The idle mode allows the software to halt the CPU when not in use while continuing to monitor interrupt service requests. The CPU clock is stopped and the entire processor context is preserved. When a interrupt occurs the processor switches back to 'run' mode and continues operating exactly where it left. The 'sleep' mode offers greatest power savings and minimum functionality. Power supply is cut off to a majority of circuits and the sleep state machine watches for a pre-programmed wakeup event. Similarly, a Bluetooth radio has four different power consumption modes - 'active', 'hold', 'sniff' and 'park' modes[1].

It is clear from the above discussion that most devices support multiple power down modes offering different levels of power consumption and functionality. An embedded system with multiple such devices can have a set of power states based on various combinations of device power states. In this chapter we outline a shutdown scheme that characterizes a system into a set of power states. The corresponding shutdown algorithm results in better power savings and enables fine grained energy-quality trade-offs.

---

1. In 'active' mode, the Bluetooth device actively participates on the wireless channel. The 'hold' mode supports synchronous packets but not asynchronous packets. This mode enables the unit to free time in order to accomplish other tasks involving page or inquiry scans. The next reduced power mode is 'sniff' mode, which basically reduces the duty cycle of the slave's listening activity. The last mode is 'park' mode, which allows a unit to not actively participate in the channel but to remain synchronized to the channel and to listen for broadcast messages. For more details on various bluetooth modes the reader is referred to [53].

4.2.1 Advanced Configuration and Power Management Interface

There exists an open interface specification called the *Advanced Configuration and Power Management Interface* (ACPI), jointly promoted by Intel, Microsoft and Toshiba [54] which standardizes how the operating system can interface with devices characterized by multiple power states to provide dynamic power management. ACPI supports a finite state model for system resources and specifies the hardware/software interface that should be used to control them. ACPI controls the power consumption of the whole system as well as the power state of each device. An ACPI compliant system has five global states. `SystemStateS0` (working state), and `SystemStateS1` to `SystemStateS4` corresponding to four different levels of sleep states. Similarly, an ACPI compliant device has four states, `PowerDeviceD0` (the working state) and `PowerDeviceD1` to `PowerDeviceD3`. The sleep states are differentiated by the power consumed, the overhead required in going to sleep and the wakeup time. In general, the deeper the sleep state, the lesser the power consumption, and the longer the wakeup time. Figure 4-1 shows the interface specification for ACPI. The Power Manager, which is a part of the OS, uses the ACPI drivers to perform intelligent shutdown.

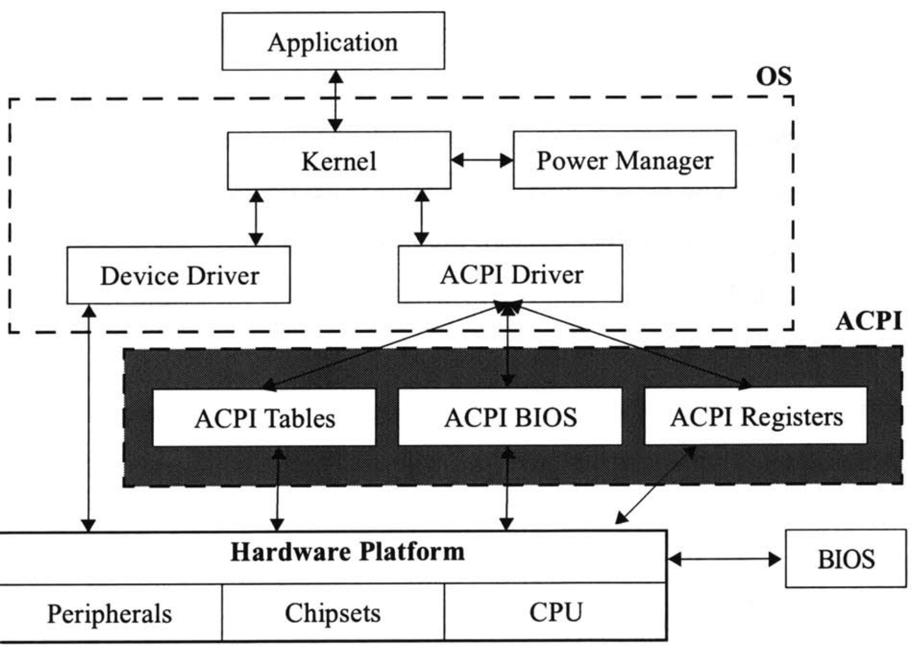


Figure 4-1: ACPI interface specification on the PC

ACPI provides low-level interfaces that allow the Operating System Power Manager (OSPM) to manage the device and system power modes. It is an enabling interface standard with the management policy is implemented in the OS itself (Power Manager block in Figure 4-1). ACPI is a PC standard and such an elaborate interface is not needed for simpler systems. A sufficient "power-aware" system model should differentiate meaningful power modes for the system and define a shutdown strategy that maximizes energy savings. The rest of this chapter describes the power manager policy for a sensor node. First, a "power-aware" sensor node model is introduced which enables the embedded operating system to make transitions to different sleep states based on observed event statistics. The adaptive shutdown algorithm is based on a stochastic analysis and renders desired energy-quality scalability at the cost of latency and missed events. Although the shutdown scheme is not ACPI compatible, the multiple sleep state formulation is along the lines of what the industry is proposing for advanced power management in PCs.

4.3 Sensor System Models

4.3.1 Sensor Network and Node Model

The fundamental idea in distributed sensor applications is to incorporate sufficient processing power in each node such that they are self-configuring and adaptive. Figure 4-2 illustrates the basic sensor node architecture. Each node consists of the embedded sensor, A/D converter, a processor with memory (which in our case will be the StrongARM SA-1100 processor) and the RF circuits. Each of these components are controlled by the micro Operating System (μ-OS) through micro device drivers. An important function of the μ-OS is to enable Power Management (PM). Based on event statistics, the μ-OS decides which devices to turn off/on.

Our network essentially consists of η homogeneous sensor nodes distributed over a rectangular region R with dimensions WxL with each node having a visibility radius of ρ. Three different communication models can be used for such a network. (i) Direct transmission (every node directly transmits to a basestation), (ii) Multi-hop (data is routed through the individual nodes towards a basestation) and (iii) Clustering. It is likely that sensors in local clusters share highly correlated data. If the distance between the neighboring sensors is less than the average distance between the sensors and the user or the bases-

tation, transmission power can be saved if the sensors collaborate locally[1]. Some of the nodes elect themselves as 'cluster heads' (as depicted by nodes in black) and the remaining nodes join one of the clusters based on a minimum transmit power criteria. The cluster head then aggregates and transmits the data from the other nodes in the cluster. Such application specific network protocols for wireless microsensor networks have been developed in [55]. It has been demonstrated that a such a clustering scheme, under certain circumstances, is an order of magnitude more energy efficient than a simple direct transmission scheme.

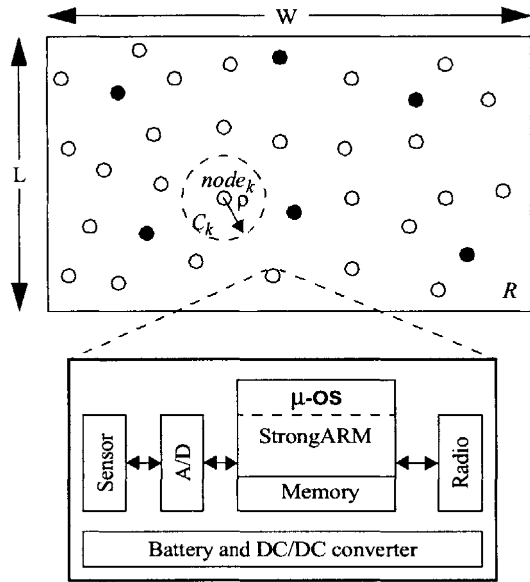


Figure 4-2: Sensor network and node architecture

4.3.2 Power Aware Sensor Node Model

A power aware sensor node model essentially describes the power consumption in different levels of node-sleep state. Every component in the node can have different power modes, e.g., the StrongARM can be in active, idle or sleep mode; the radio can be in transmit, receive, standby or off mode. Each node-sleep state corresponds to a particular combination of component power modes. In general, if there are N components labelled (1, 2, ..., N), each with k_i number of sleep states, the total number of node-sleep states are $\prod k_i$.

1. Under good conditions, radio transmission power increases quadratically with transmission distance.

Every component power mode is associated with a latency and energy overhead for transitioning to that mode. Therefore each node sleep mode is characterized by an energy consumption and a latency overhead. However, from a practical point of view not all the sleep states are useful.

Table 4-1 enumerates the component power modes corresponding to 5 different useful sleep states for the sensor node. Each of these node-sleep modes correspond to an increasingly deeper sleep state and is therefore characterized by an increasing latency and decreasing power consumption. These sleep states are chosen based on working conditions of the sensor node, e.g., it does not make sense to have the memory in the active state and everything else completely off. State s_1 is the completely "active" state of the node where it can sense, process, transmit and receive data. In state s_1, the node is in a "sense & receive" mode while the processor is on standby. State s_2 is similar to state s_1 except that the processor is powered down and is woken up when the sensor or the radio receives data. State s_3 is the "sense only" mode where everything except the sensing front-end is off. Finally, state s_4 represents the completely off state of the device. The design problem is to formulate a policy of transitioning between states based on observed events so as to maximize energy efficiency. It can be seen that the power aware sensor model is similar to the system power model in the ACPI standard. The sleep states are differentiated by the power consumed, the overhead required in going to sleep and the wakeup time. In general, the deeper the sleep state, the lesser the power consumption, and the longer the wakeup time.

Table 4-1: Useful sleep states for the sensor node

State	StrongARM	Memory	Sensor, A/D	Radio
s_0	active	active	on	tx, rx
s_1	idle	sleep	on	rx
s_2	sleep	sleep	on	rx
s_3	sleep	sleep	on	off
s_4	sleep	sleep	off	off

4.3.3 Event Generation Model

An event is said to occur when a sensor node picks up a signal with power above a pre-defined threshold. For analytical tractability we assume that every node has a uniform radius of visibility, ρ. In real applications the terrain might influence the visibility radius. An event can be static (e.g., a localized change in temperature/pressure in an environment monitoring application) or can propagate (e.g., signals generated by a moving object in a tracking application). In general, events have a characterizable (possibly non-stationary) distribution in space and time. We will assume that the temporal behavior of events over the entire sensing region, R, is a Poisson process with an average rate of events given by λ_{tot} [56]. In addition, we assume that the spatial distribution of events is characterized by an independent probability distribution given by $p_{XY}(x,y)$. Let p_{ek} denote the probability that an event is detected by $node_k$, given the fact that it occurred in R.

$$p_{ek} = \frac{\int_{C_k} p_{XY}(x, y)dxdy}{\int_{R} p_{XY}(x, y)dxdy} \tag{4-1}$$

Let $p_k(t,\ n)$ denote the probability that n events occur in time t at $node_k$. Therefore, the probability of no events occurring in the region C_k (the visibile area of $node_k$), over a threshold interval T_{th}, is given by

$$\begin{aligned} p_k(T_{th}, 0) &= \sum_{i=0}^{\infty} \frac{e^{-\lambda_{tot}T_{th}}(\lambda_{tot}T_{th})^i}{i!}(1-p_{ek})^i \\ &= e^{-p_{ek}\lambda_{tot}T_{th}} \end{aligned} \tag{4-2}$$

Let $p_{th,k}(t)$ be the probability that at least one event occurs in time t at $node_k$.

$$p_{th,k}(T_{th}) = 1 - p_k(T_{th}, 0) = 1 - e^{-p_{ek}\lambda_{tot}T_{th}} \tag{4-3}$$

i.e., the probability of at least one event occurring is an exponential distribution characterized by a spatially weighted event arrival rate $\lambda_k = \lambda_{tot}\,p_{ek}$.

In addition, to capture the possibility that an event might propagate in space we describe each event by a position vector, $\bar{\mathbf{p}} = \bar{\mathbf{p}}_0 + \int \bar{v}(t)dt$. Where $\bar{\mathbf{p}}_0$ is the coordinates of

the point of origin of the event and $\bar{v}(t)$ characterizes the propagation velocity of the event. The point of origin has a spatial and temporal distribution described by Equation 4-1 to Equation 4-3. We have analyzed three distinct classes of events: (i) $\bar{v}(t) = 0$, the events occur as stationary points, (ii) $\bar{v}(t) = const$, the event propagates with fixed velocity (e.g., a moving vehicle), and, (iii) $| \bar{v}(t) | = const$, the event propagates with fixed speed but random direction (i.e., a random walk).

4.4 Shutdown Policy

4.4.1 Sleep State Transition Policy

Assume an event is detected by $node_k$ at some time and it finishes processing it at t_1 and the next event occurs at time $t_2 = t_1 + t_i$. At time t_1, $node_k$ decides to transition to a sleep state s_k from the active state s_0 as shown in Figure 4-3. Each state s_k has a power consumption P_k, and the transition time to it from the active state and back is given by $\tau_{d,k}$ and $\tau_{u,k}$ respectively. By our definition of node-sleep states, $P_j > P_i$, $\tau_{d,i} > \tau_{d,j}$ and $\tau_{u,i} > \tau_{u,j}$ for any $i > j$. The power consumption between the sleep modes is modeled as a linear ramp between the states. When the node transitions from state s_1, to say, state s_k, individual components such as the radio, memory, and processor are progressively powered down. This results in a stepped variation in power consumption between the states. The linear ramp is analytically simpler to handle and approximates the process reasonably well.

We will now derive a set of sleep time thresholds $\{ T_{th,k} \}$ corresponding to the states $\{ s_k \}$, $0 \le k \le N$ (for N sleep states) such that transitioning to a sleep state s_k from state s_0 will result in a net energy loss if the idle time $t_i < T_{th,k}$ because of the transition energy overhead. This assumes that no productive work can be done in the transition period [39], which is usually true, e.g., when a processor wakes up the transition time is the time required for the PLLs to lock, the clock to stabilize and the processor context to be restored. The energy savings because of state transition is given by the difference in the area under the graphs shown in Figure 4-3.

$$\begin{aligned} E_{save,k} &= P_0 t_i - \left(\frac{P_0 + P_k}{2}\right)(\tau_{d,k} + \tau_{u,k}) - P_k(t_i - \tau_{d,k}) \\ &= (P_0 - P_k)t_i - \left(\frac{P_0 - P_k}{2}\right)\tau_{d,k} - \left(\frac{P_0 + P_k}{2}\right)\tau_{u,k} \end{aligned} \tag{4-4}$$

and such a transition is only justified when $E_{save,k} > 0$. This leads us to the threshold

$$T_{th,k} = \frac{1}{2}\left[\tau_{d,k} + \left(\frac{P_0 + P_k}{P_0 - P_k}\right)\tau_{u,k}\right] \quad (4\text{-}5)$$

which implies that the longer the delay overhead of the transition $s_0 \rightarrow s_k$, the higher the energy gain threshold, and the more the difference between P_0 and P_k, the smaller the threshold.

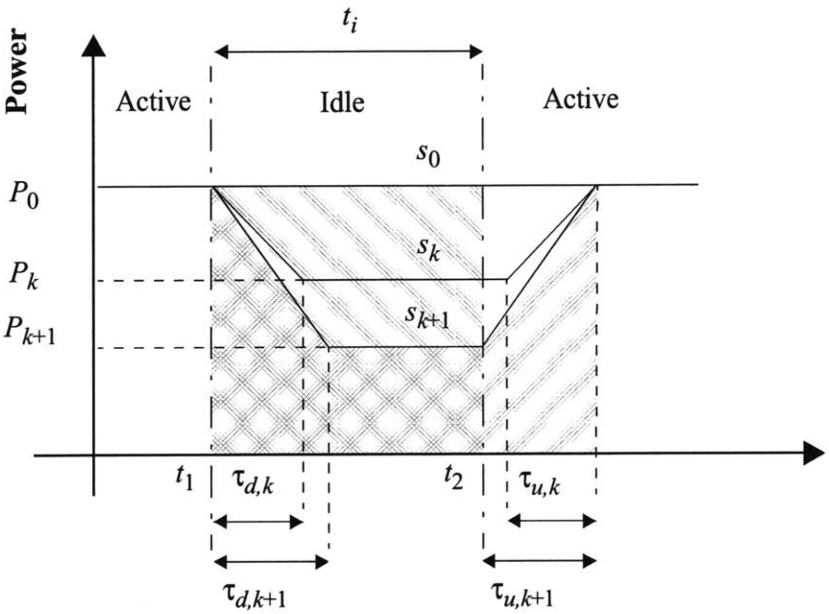


Figure 4-3: State transition latency and power

Table 4-2 lists the power consumption of a sensor-node described in Figure 4-2 in the different power modes and the corresponding energy gain thresholds. Since the node consists of off the shelf components, it is not optimized for power consumption. However, we will use the threshold and power consumption numbers detailed in Table 4-2 to illustrate our basic idea. The steady state shutdown algorithm is as follows

```
if( eventOccurred() == true ) {
   processEvent();
   ++eventCount;
   lambda_k = eventCount/getTimeElapsed();
      for( k=4; k>0; k-- )
         if( computePth( Tth(k) ) < pth0 )
            sleepState(k);
}
```

Table 4-2: Sleep state power, latency and threshold

State	P_k (mW)	τ_k (ms)	$T_{th,k}$
s_0	1040	-	-
s_1	400	5	8
s_2	270	15	20
s_3	200	20	25
s_4	10	50	50

When an event is detected at $node_k$, it wakes up and processes the event. It then updates a global `eventCount` counter that stores the total number of events registered by $node_k$. The average arrival rate, λ_k, for $node_k$ is then updated. This requires use of an μ-OS timer based system function call `getTimeElapsed()` which returns the time elapsed since the node was turned on. The μ-OS then tries to put the node into sleep state s_k (starting from the deepest state s_4 through s_1) by testing the probability of an event occurring in the corresponding sleep time threshold $T_{th,k}$ against a system defined constant, p_{th0}.

4.4.2 Missed Events

All the sleep states, except state s_4 have the sensor and A/D circuit on. Therefore if an event is detected (i.e., the signal power is above a threshold level) the node transitions to state s_0 and processes the event. The only overhead involved is latency (worst case being about 25 ms). However, in state s_4, the node is almost completely off and it must decide on its own when to wake up. In sparse event sensing systems (e.g., vehicle tracking, seismic detection, etc.) the inter-arrival time for events is much greater than the sleep time thresholds, $T_{th,k}$. Therefore, the sensor node will invariably go into the deepest sleep state, s_4. The processor must watch for pre-programed wake-up signals. These signal conditions are programmed by the CPU prior to entering the sleep state. To be able to wake up on its own the node must be able to predict the arrival of the next event. An optimistic prediction

might result in the node waking up unnecessarily while a pessimistic strategy will result in some events being missed.

Being in state s_4 results in missed events as the node has no way of knowing if anything significant occurred. What strategy gets used is a pure design concern based on how critical the sensing task is. We discuss two possible approaches.

- *Completely disallow* s_4 - If the sensing task is critical and there exists an event that cannot be missed, this state must be disabled.
- *Selectively disallow* s_4 - This technique can be used if events are spatially distributed and not totally critical. Both random and deterministic approaches can be used. In the protocol described in [55] the 'cluster heads' can have a disallowed s_4 state while the normal nodes can transition to s_4. Alternatively, the scheme that we propose is more homogeneous. Every $node_k$ that satisfies the sleep threshold condition for s_4 goes to sleep with a system defined probability p_{s4} for a time duration given by

$$t_{s4,\,k} = -\frac{1}{\lambda_k}\ln(p_{s4}) \qquad (4\text{-}6)$$

Equation 4-6 describes the steady state behavior of the node and the sleep time is computed such that the probability that no events occur in $t_{s4,\,k}$, i.e., $p_k(t_{s4,\,k}, 0) = p_{s4}$. However, when the sensor network is switched on and no events have occurred for a while, λ_k is zero. To account for this we disallow transition to state s_4 until at least one event is detected. We can also have an adaptive transition probability, p_{s4}, that is almost zero initially and increases as events are detected later on. The probabilistic state transition is described in Figure 4-4.

The advantage of the algorithm is that efficient energy trade-offs can be made with event detection probability. By increasing p_{s4}, the system energy consumption can be reduced while the probability of missed events will increase and vice versa. Our overall shutdown policy is governed by two implementation specific probability parameters, (i) p_{th0} and (ii) p_{s4}.

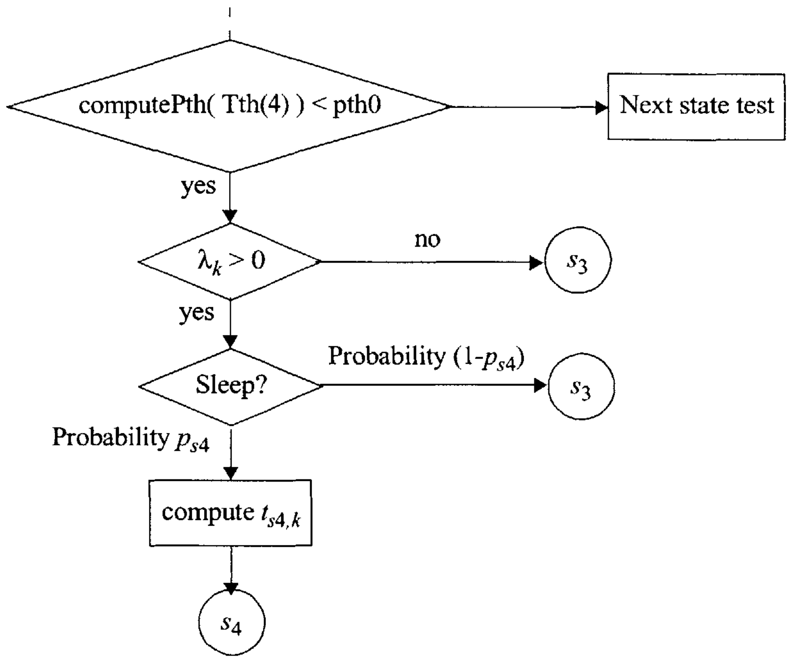


Figure 4-4: Transition algorithm to 'almost off' s_4 state

4.5 Java Based Event Driven Shutdown Simulator

The shutdown algorithm along with a behavioral model of the sensor node, sensor network and event generator was implemented in Java to simulate the energy efficiency of the system. The sensor node object captures the "power-aware" description of the sensor node. Parameters such as state descriptions ($\{s_k, P_k, \tau_k\}$) are provided in a separate file (`simProperties.dat`) that can be modified and loaded at run-time. This file also defines the threshold parameters ($T_{th,k}, p_{th0}$) that the Power Manager in the sensor node object uses to make shutdown policy decision as discussed in the previous sections of this chapter. In addition, network parameters such as distribution area (W, L) and visibility radius (ρ_k) are also incorporated into the file.

The simulator defines an event generator object which can be used to fire events at the network. The statistical properties of the event generator can be altered using the simulation parameters file (e.g., Gaussian and Poisson parameters, etc.). The simulation methodology is discrete. A timestep parameter determines the simulation granularity. For sparse events, a larger timestep can be used for faster simulation (e.g., an earthquake sensor

might register events in months!). The energy consumption output is a normalized estimation comparing the energy availability among different nodes after the simulation. Actual energy estimation can easily be estimated by scaling with data from one real node. The overall broad simulation framework is captured in Figure 4-5.

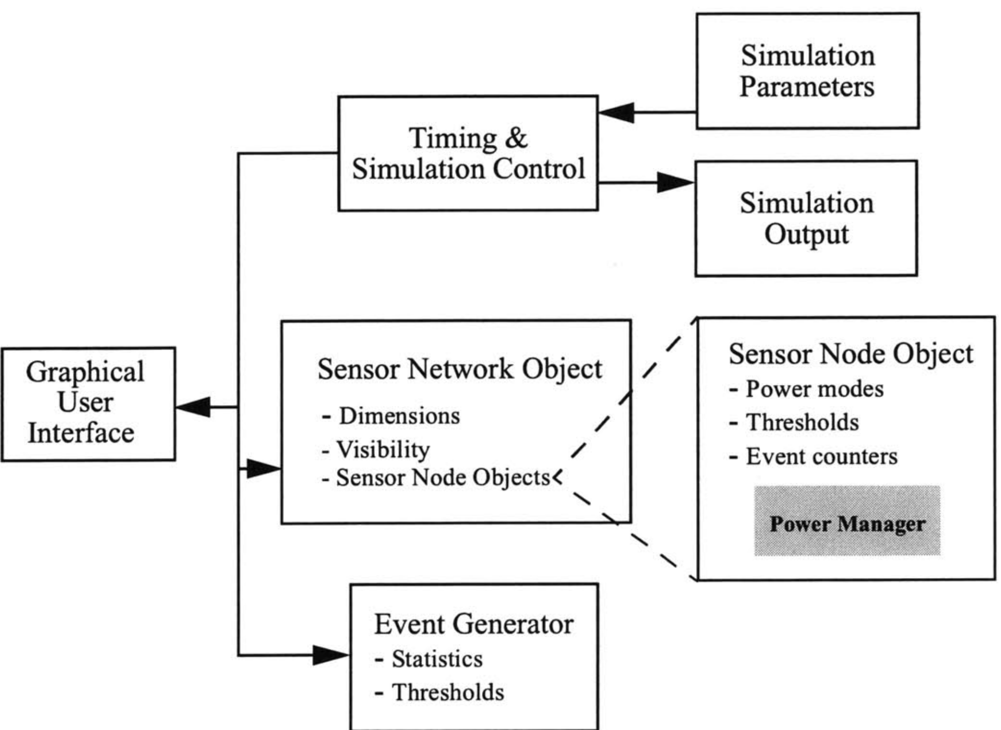


Figure 4-5: Java based event driven sensor network shutdown simulator

4.5.1 Results

An intermediate simulation snapshot of a $\eta = 1000$ node system distributed uniformly and randomly over a 100 m x 100 m area is shown in Figure 4-6. Color codes are used to show the state of a particular sensor. The events in this case are concentrated as shown. It is interesting to note that although some nodes in the periphery of the event zone have gone into the deepest sleep state s_4 (after about 4 seconds), a substantial number of nodes even further beyond are still not allowed to enter s_4, since they are still waiting for the first few events to get calibrated. The visibility radius of each sensor was assumed to be $\rho = 10$ m. The sleep state thresholds and power consumption are shown in Table 4-2.

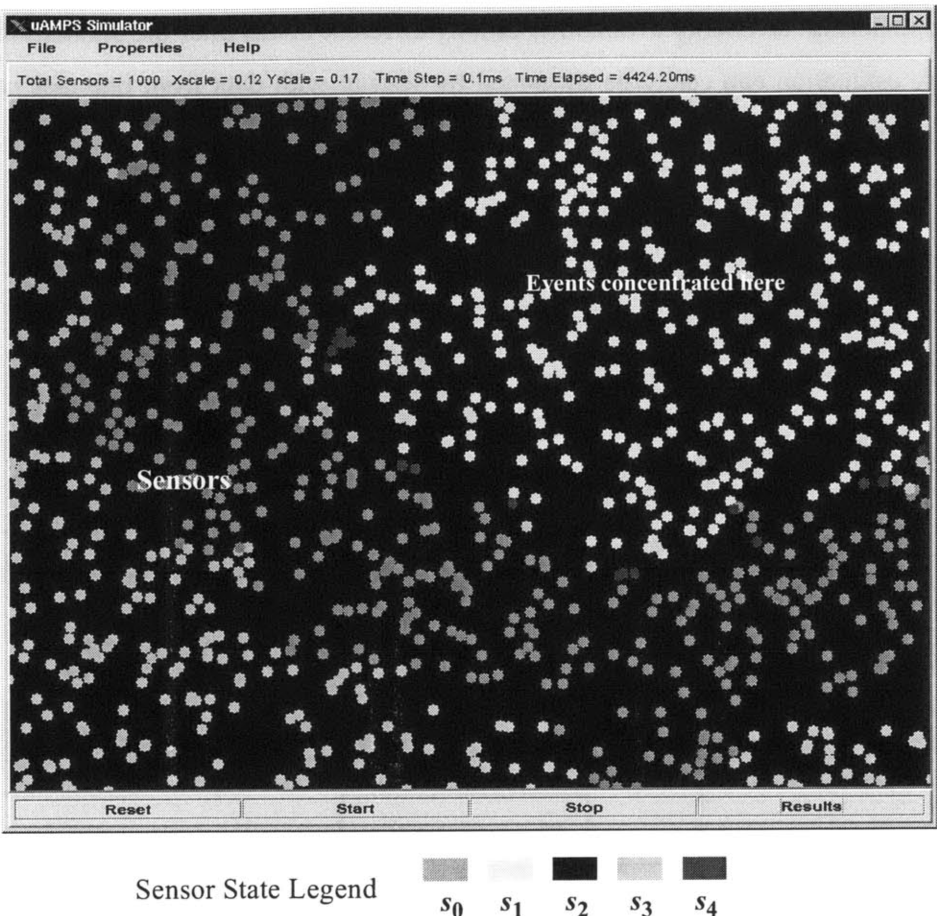


Figure 4-6: Java based event driven sensor network shutdown simulator

Figure 4-7 shows the overall spatial node energy consumption for events with a Gaussian spatial distribution centered around (25, 75). The interarrival process is Poisson with $\lambda_{tot} = 500\ s^{-1}$. It can be seen that the node energy consumption tracks the event probability. In the non-power managed scenario, we would have a uniform energy consumption in all the nodes.

One drawback of the whole scheme is that there is a finite and small window of interarrival rates λ_{tot} over which the fine grained sleep states can be utilized. In general, the more differentiated the power states (i.e., the greater the difference in their energy and latency overheads) the wider the range of interarrival times over which all sleep states can be utilized. Figure 4-8 shows the range of event arrival rates at a node (λ_k) over which the states s_1 - s_3 are used significantly. If $\lambda_k < 13.9\ s^{-1}$, transition to state s_4 is always possible

(i.e., at least the threshold condition is met, actual transition, of course, occurs with probability p_{s4}). Similarly, if $\lambda_k > 86.9$ s^{-1}, the node must always be in the most active state. These limits have been computed using the nominal $p_{th0} = 0.5$. Using a higher value of p_{th0} would result in frequent transitions to the sleep states and if events occur fast enough this would result in increased energy dissipation associated with the wake-up energy cost. A smaller value of p_{th0} would result in a pessimistic scheme for sleep state transition and therefore lesser energy savings.

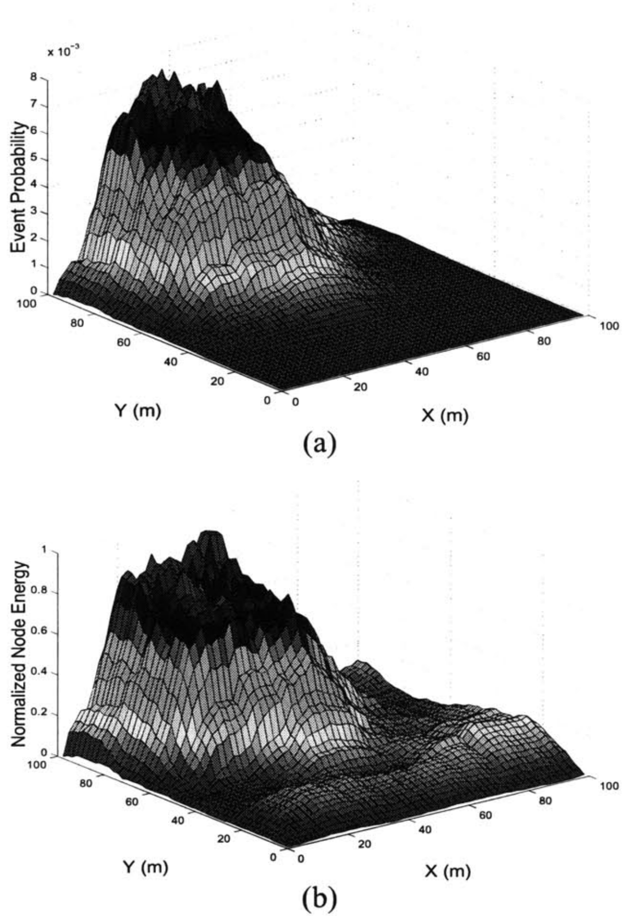


Figure 4-7: (a) Spatial distribution of events (Gaussian) and (b) Spatial energy consumption in the sensor nodes

Figure 4-9 illustrates the Energy-Quality trade-off of our shutdown algorithm. By increasing the probability of transition to state s_4 (i.e., increasing p_{s4}) energy can be saved at the cost of increased possibility of missing an event. Such a graceful degradation of quality with energy is highly desirable in energy constrained systems.

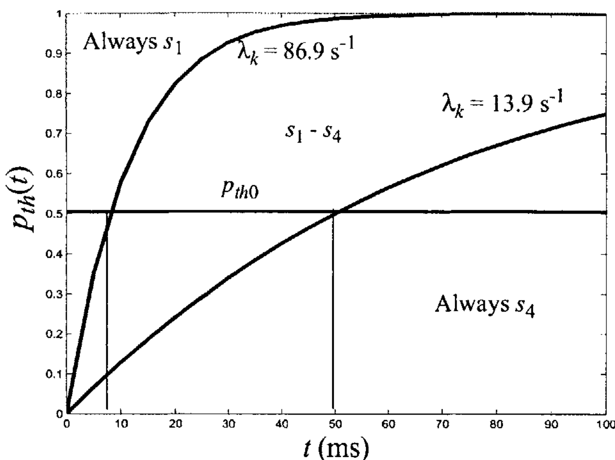


Figure 4-8: Event arrival rates at a node and corresponding limits on state utilization

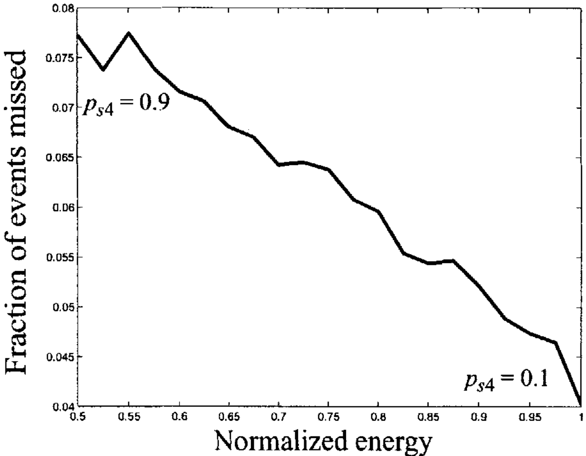


Figure 4-9: Fraction of events missed versus energy consumption

4.6 Summary of Contributions

In this chapter, we proposed and simulated an operating system based shutdown scheme for idle power management using a sensor node as an example. The scheme explicitly characterizes the meaningful power states of a node and uses a probabilistic technique to make transitions to the low power modes based on observed event statistics. The scheme is simple to implement and has negligible memory overhead. The technique we proposed is fairly general and can be used for power management in any system characterized by different levels of power consumption in various stages of shutdown.

The key contribution in this chapter is that we developed the shutdown algorithm by explicitly accounting for *multiple* sleep states. We showed that simply transitioning to the deepest state is not optimal because of the finite power and latency overheads associated with a sleep state and the loss in quality. We derived time thresholds for each state, as a function of the power and latency for those states, before which no energy saving can result from transitioning to that state and getting active again. We also demonstrated the feasibility of a graceful energy-quality trade-off using our shutdown scheme, which is desirable in energy constrained systems. These ideas have been tested in an event driven simulator written in Java.

Chapter 5
System Implementation

In the previous chapters we described various operating system directed power management techniques and the theoretical/simulated energy savings possible from them. While each scheme individually sounded quite promising, it's system level implication was unclear. For example, although DVS might result in 50% processor energy savings, the processor might account for only 30% of the power budget in a system. This chapter attempts to quantify actual system level energy savings possible by exploiting some of the active and idle power management mechanisms discussed before. We have used the μAMPS sensor node as our target system. Although the actual energy numbers might vary significantly from one system to another, the sensor node example will demonstrate the overall efficacy of our proposed power management techniques and give the reader a flavor of what type of energy savings to expect if similar techniques are used in other systems.

This chapter also provides a description of the operating system that has been developed for the μAMPS sensor node incorporating some of the power management features discussed in the previous chapters. Instead of building an entire OS from scratch we built the μAMPS OS using an existing, real-time, embedded, open source OS available from Redhat, Inc. [59]. In essence, we ported the operating system to our sensor hardware target and added an entire power management layer and a corresponding Application Programming Interface (API) on top of it. We begin this chapter with a discussion of some standard embedded operating systems and their features and the motivation for choosing Redhat's eCos (Embedded Configurable Operating System). This is followed by a brief discussion of the sensor node hardware and the various power management hooks that have been incorporated into its design. We then describe the OS architecture and the power management API. Finally system level energy savings derived from operating system directed power management are quantified.

5.1 Embedded Operating Systems Overview

A variety of embedded operating systems exist. In this section, we describe the two most popular ones and also explain why eCos was chosen as our base operating system.

5.1.1 Windows CE

Microsoft Windows CE [60] is a scaled-down operating system designed specifically for what Microsoft terms "information appliances". In Microsoft's vision, these appliances range from handheld computing devices to mobile phones to automobiles to industrial equipment. The primary feature that differentiates Windows CE from competitors such as the Palm OS is that CE is a 32-bit, multi-threaded, multi-tasking operating system. Again, the design philosophy differences show up here. Palm Computing designers opted for a simpler, power-conserving operating system since they were assuming that the handheld user only wanted to retrieve simple information quickly and efficiently. CE designers, on the other hand, opted for a more powerful solution that offers the potential of running processor-intensive applications (such as MP3 and video playback, spreadsheet calculations, etc.). Benchmarks have shown the Windows CE operating system in combination with MIPS and Hitachi's SH hardware to be far superior in performance to the Palm platform based on the Motorola Dragonball processor (essentially an older 68000 variant). Obviously, there are a lot of factors to consider when choosing an operating system. What sort of operations will be performed on a routine basis? Also, what type of battery life is expected? Batteries on Palm devices can last on the order of six weeks; CE color devices, on the other hand, can drain batteries within a single day!

Power Management

Because Windows CE was designed to be portable to a wide range of processors, power management details differ from one device to the next. However, the CE API does support a set of power monitoring functions in order to allow applications to determine the remaining life of the battery, whether batteries are currently being used, and whether the batteries are currently being charged (i.e., is the device plugged into AC power). Well-written CE applications will monitor power levels and will alert users and gracefully exit as levels become critically low.

Pros and Cons

Windows CE's greatest advantage is easily its similarity to the other members of the Microsoft Windows family of operating systems. A vast majority of professional software developers have experience developing applications using the Windows API with Microsoft tools such as Visual C++ and Visual Basic. A number of factors can be attributed to CE's lack of widespread success: CE's extra memory requirements, user interface, and battery usage requirements lead that list. The unavailability of the source code is another reason for its lack of popularity with new experimental systems like μAMPS. In addition, the μAMPS system has much simpler requirements compared to what CE can provide and, therefore, the overheads cannot be justified.

5.1.2 Palm OS

As opposed to Windows CE, Palm OS [61] is not a multi-tasking operating system. PalmOS and Windows CE machines are designed from two completely different approaches, and the kernels of the two operating systems reflect this. Perhaps the main outstanding difference is that the Palm OS (based on the AMX kernel from embedded systems vendor KADAK) supports and is optimized for a very specific hardware reference platform designed entirely by Palm Computing. Because of this, there is little deviation between the Palm OS platform vendors as far as hardware differences go. Microsoft Windows CE is designed to support a much wider range of hardware. There are no fewer than four different versions of the Windows CE kernel for different CPU's (NEC MIPS and its variants, Intel/AMD X86, Hitachi SH-3 and SH-4, Intel StrongARM) along with other vendor-specific versions of Windows CE to handle different kinds of screens, keyboards, modems and peripheral devices. It is fair to say that there is only one version of Palm OS, whereas Windows CE is compiled specifically for the machine that it is being designed to run on. Nevertheless, the success of Palm OS is largely because of its simplicity.

Power Management

The Palm OS and its underlying hardware support three modes of operations: sleep mode, doze mode, and running mode. In sleep mode, the device appears to be off although the real-time clock and interrupt generation circuitry are still running. Sleep mode is

entered when the device remains inactive for a pre-defined time or when the user presses the 'Off' button. In doze mode, the system appears to be on (the display remains active and the processor's clock is running), however no instructions are being processed. The device is continually in this mode while it is on as long as it has no user input to process. Run mode is entered when there is a user event to process. The device reenters doze mode immediately after processing the final event. System API calls allow applications to enter doze mode as well as determine current time-out values and battery levels.

Pros and Cons

Because the OS was built from the ground up to run on small, low powered devices, it is optimized and has low memory requirements. The OS is based on an event driven programming model and, as such, is very conducive for idle power management. Its primary disadvantage is the lack of flexibility. Palm devices currently use the Motorola 68328 (Dragonball) processor which is limited in terms of speed and computation capabilities. The addressable memory space is also limited (less than 12MB for PalmOS 3.x). The platform restrictions of PalmOS, coupled with the fact that like CE it is designed for more user interface driven applications, also made it unsuitable for the μAMPS node.

5.1.3 Redhat eCos

Redhat eCos is an open source, real-time operating system for deeply embedded applications. It meets the requirements of the embedded space that Linux cannot yet reach. Linux currently scales from a minimal size of around 500 KB of kernel and 1.5 MB of RAM, all before taking into consideration application and service requirements. eCos can provides the basic runtime infrastructure necessary to support devices with memory footprints in the tens to hundreds of KB, with real-time requirements. Some of the key features that made eCos our base operating system are:

- *Scalability* - eCos has over 200 configuration options (which can be chosen using a handy configuration tool) for fine grain scalability, and code size can be as small as a few kilobytes.
- *Compatibility* - eCos has μITRON [62] compatibility and also supports EL/IX [63] Level I, a Linux compatibility interface.

- *Multi-platform* - eCos can be configured and built to target a variety of processors and platforms (e.g., ARM, IA32, Matsushita AM3x, MIPS, NEC V8xx, PowerPC, SPARC, SuperH etc.)
- *Modularity* - eCos is implemented with a Hardware Abstraction Layer (HAL) that makes it easier to port to new platforms. It is also implemented in such a way that makes it easy to plug in a custom scheduler, device driver, etc.
- *Open Source* - eCos source code can be freely downloaded from [64].
- *Development and Support* - eCos uses the standard GNU [65] toolchain. There is an active mailing list (ecos-discuss) for free support.

Power Management

The original eCos source does not contain any specific power management techniques. Later in this chapter, we will describe how eCos has been ported to incorporate sophisticated active and idle power management schemes on the μAMPS sensor node.

5.2 Sensor Hardware

Figure 5-1 shows the current (version 1) processor board of the μAMPS sensor node. It is based on the StrongARM SA-1110 processor and has 1 MB of on board SRAM and Flash memory. The board runs at a nominal battery (single lithium primary cell) power supply of about 4.0 V. The on board power supply circuits generate a 3.3 V supply for all digital circuits. A separate analog power supply is also generated to isolate the digital power supply noise from the analog circuits. The 3.3 V digital power supply also powers the I/O pads of the StrongARM SA-1110 processor. The core power supply is generated through a DVS circuit that can regulate the power supply from 0.925 V to a maximum of 2.0 V at a very good conversion efficiency of about 85%. The radio module (which is still under development), is on a similar sized board and consists of a dual power 2.4 GHz radio for 10 m and 100 m ranges. The 16 bit bus interface connector will allow the radio module to be stacked onto the processor board. In addition, the connector allows a different sensor board (e.g., a seismic sensor) to be stacked as well. The processor board also has an RS-232 and a USB connector for remote debugging and connecting to a basestation. The board features a built in acoustic sensor (a microphone, some opamps and A/D

circuit) that talk to the StrongARM processor using the Synchronous Serial Port (SSP). The opamp gains are programmable and processor controlled. An envelop detect mechanism has also been incorporated into the sensor circuit which bypasses the A/D circuit and wakes up the processor when the signal energy crosses a certain programmable threshold. Using this feature can significantly reduce the power consumption in the sense mode and allows for truly event driven computation.

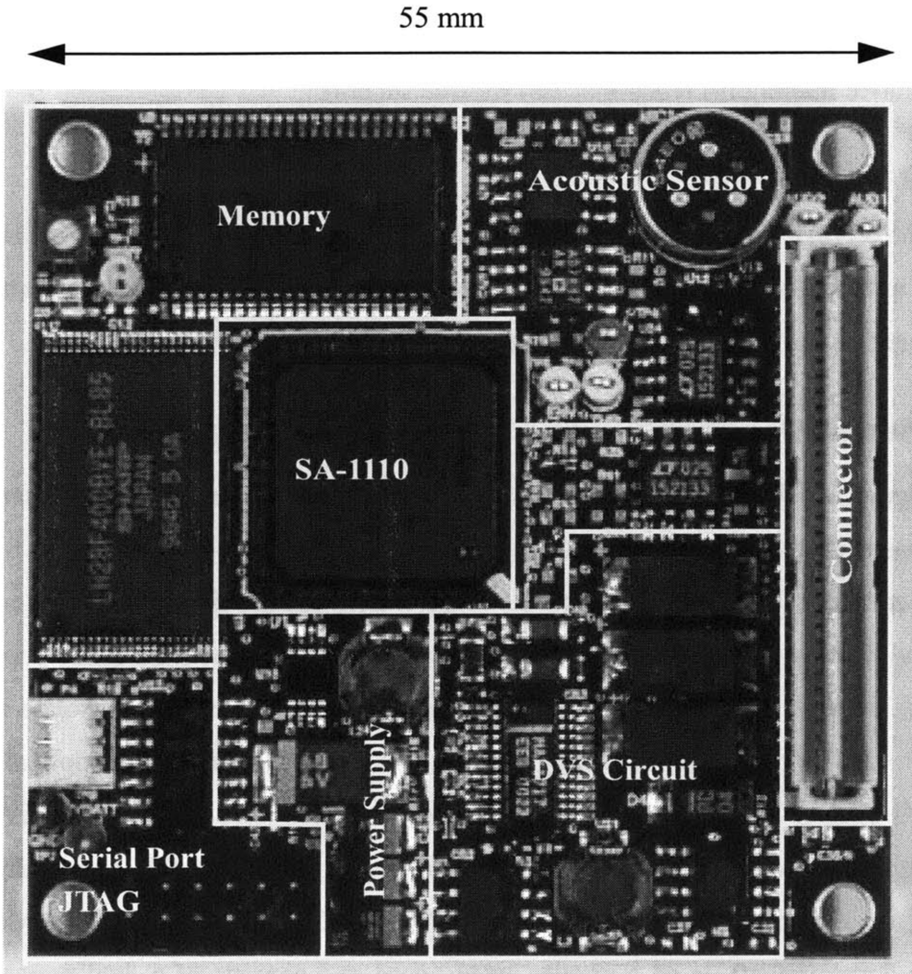


Figure 5-1: μAMPS processor board (designer: Nathan Ickes)

5.2.1 DVS Circuit

The basic variable core power supply schematic is shown in Figure 5-2. The MAX1717 step-down controller is used to regulate the core supply voltage dynamically

through the 5-bit digital-to-analog converter (DAC) inputs over a 0.925 V to 2 V range. The converter works on the following principle. A variable duty cycle Pulse Width Modulated (PWM) signal alternately turns on the power transistors M1 and M2. This produces a rectangular wave at the output of the transistors with duty cycle D. The LC low pass filter passes a desired DC output equal to $DV_{battery}$ while attenuating the AC component to an acceptable ripple. The duty cycle D is controlled using the DAC pins (D0:D4) which results in 30 voltage levels (two combinations are not allowed). A two wire remote sensing scheme compensates for voltage drops in the ground bus and output voltage rail. The StrongARM sets the DVS enable pin on the voltage regulator depending on whether DVS capability is desired or not. A feedback signal from the regulator lets the processor know if the output core voltage is stabilized. This is required for error free operation during voltage scaling.

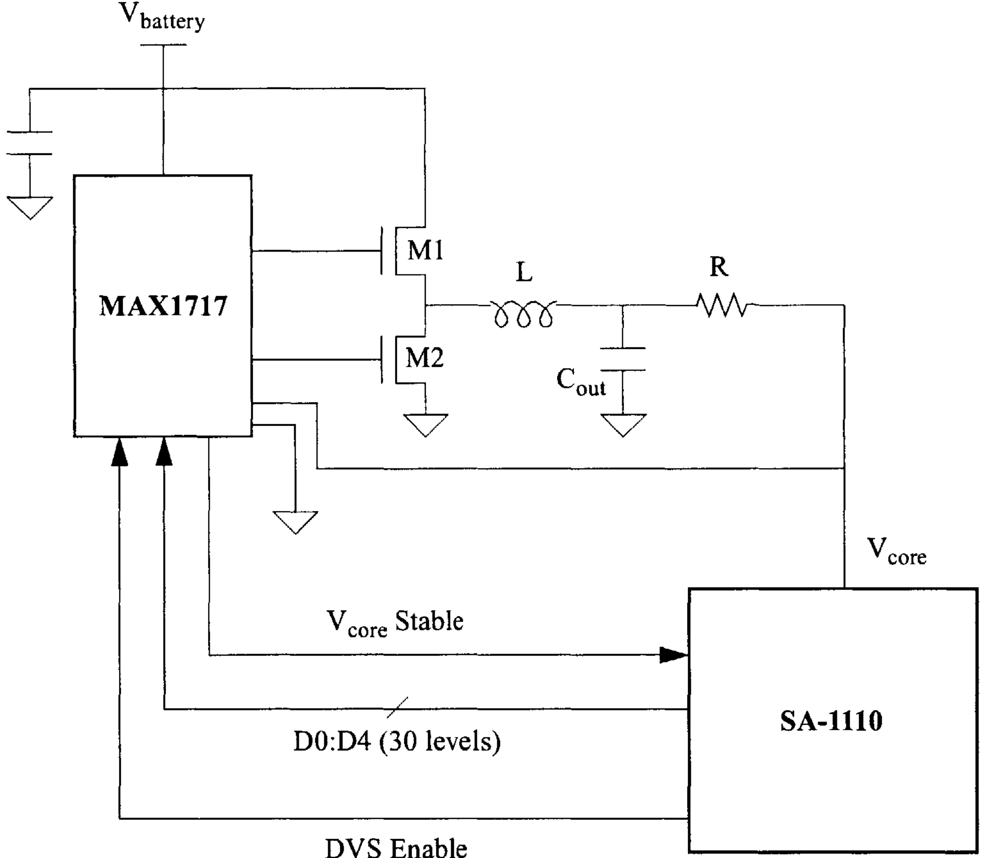


Figure 5-2: DVS circuit schematic

The processor clock frequency change involves updating the contents of the core Clock Configuration Register (CCF) of the SA-1110 [66]. The core clock is derived by multiplying the reference crystal oscillator clocks using a Phase Locked Loop (PLL). based on CCF register settings as shown in Table 5-1. The core clock (DCLK) can be driven using either the fast CCLK or the memory clock (MCLK) which runs at half the frequency of CCLK. The core clock uses CCLK normally except when waiting for fills to complete after a cache miss. Core clock switching between CCLK and MCLK can be disabled by setting a control register appropriately.

Table 5-1: SA-1110 core clock configurations and minimum core supply voltage

CCF(4:0)	Core Clock Frequency (CCLK) in MHz		Core Voltage (V) [3.6864 MHz Osc]
	3.6864 MHz Oscillator	3.5795 MHz Oscillator	
00000	59.0	57.3	1.000
00001	73.7	71.6	1.050
00010	88.5	85.9	1.125
00011	103.2	100.2	1.150
00100	118.0	114.5	1.200
00101	132.7	128.9	1.225
00110	147.5	143.2	1.250
00111	162.2	157.5	1.350
01000	176.9	171.8	1.450
01001	191.7	186.1	1.550
01010	206.4	200.5	1.650
01011	221.2	214.8	1.750
01100-11111	-	-	

The sequence of operations during a voltage and frequency update depends on whether we are increasing or decreasing the processor clock frequency as shown in Figure 5-3. When we are increasing the clock frequency we first need to increase the core supply voltage to the minimum required for that particular frequency. The optimum voltage frequency pairs are stored in a lookup table. Once the core voltage is stabilized we can pro-

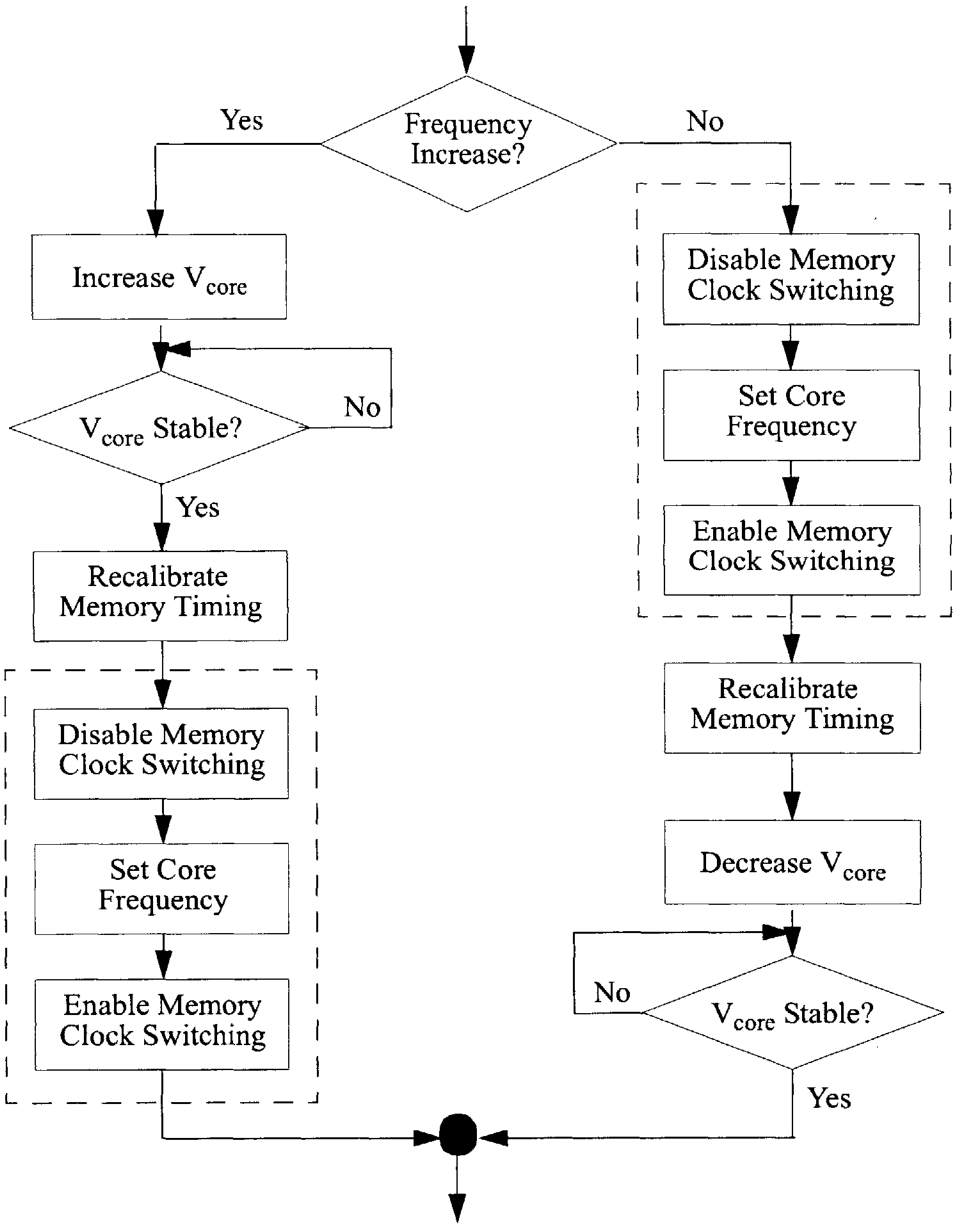


Figure 5-3: Sequence of operations during a voltage and frequency switch

ceed with the frequency update. The first step involves recalibrating the memory timings. This is done by setting an appropriate value in the MSC control register. Before we increase the core clock (CCLK) frequency we disable clock switching between CCLK and MCLK to avoid inadvertent switch of the core clock. CCLK frequency is changed by setting the CCF register. Once this is done core clock switching between CCLK and MCLK is enabled. The sequence of operations are somewhat reversed when reducing frequency.

We first update the core clock frequency (following the same 3 basic steps mentioned above). Before we can reduce the core voltage we must recalibrate the memory timing. This is required because the once the core clock frequency is reduced, memory read-write will result in errors unless the memory timing is adjusted (e.g., when reading the voltage-frequency lookup table). Subsequently, the core voltage is reduced and normal operation is started once the core voltage stabilizes. To ensure correct operation, the entire voltage frequency update has to be done is an atomic fashion. For example, if an interrupt occurs while frequency is updated and memory has not been recalibrated, execution errors might occur.

5.2.2 Idle Power Management Hooks

The sensor node has been designed to specifically allow a set of sleep states similar to the one described in the previous chapter. In addition, it has hardware support for event driven computation. The overall schematics is shown in Figure 5-4. The General Purpose I/O (GPIO) pins on the StrongARM are used to generate and receive various signals from the peripherals. The SA-1110 features 28 GPIO pins, each of which can be configured as an input or an output pin using the GPIO Pin Direction Register (GPDR). When programmed as an output, the pins can be controlled by writing to the GPIO Pin Output Set Register (GPSR) and the GPIO Pin Output Clear Register (GPCR). When configured as an input, the current state of each pin can be read off the GPIO Pin Level Register (GPLR). In addition, the GPIO pins can be configured to specifically detect a rising or falling edge. In our implementation, four GPIO pins are dedicated to power supply control in the system. The entire analog power supply can be switched off when no sensing is required. Alternately, only the power supply to the Low Pass Filter (LPF) can be switched off and the envelop energy sensing circuit could be used to trigger a signal to the processor. When this happens, the processor could enable the LPF and start reading data off the A/D converter using the SSP (Synchronous Serial Port). The signal detection threshold is also programmable using other GPIO pins. Similar power supply control is available for the radio module. The processor can turn off the radio when it is not required.

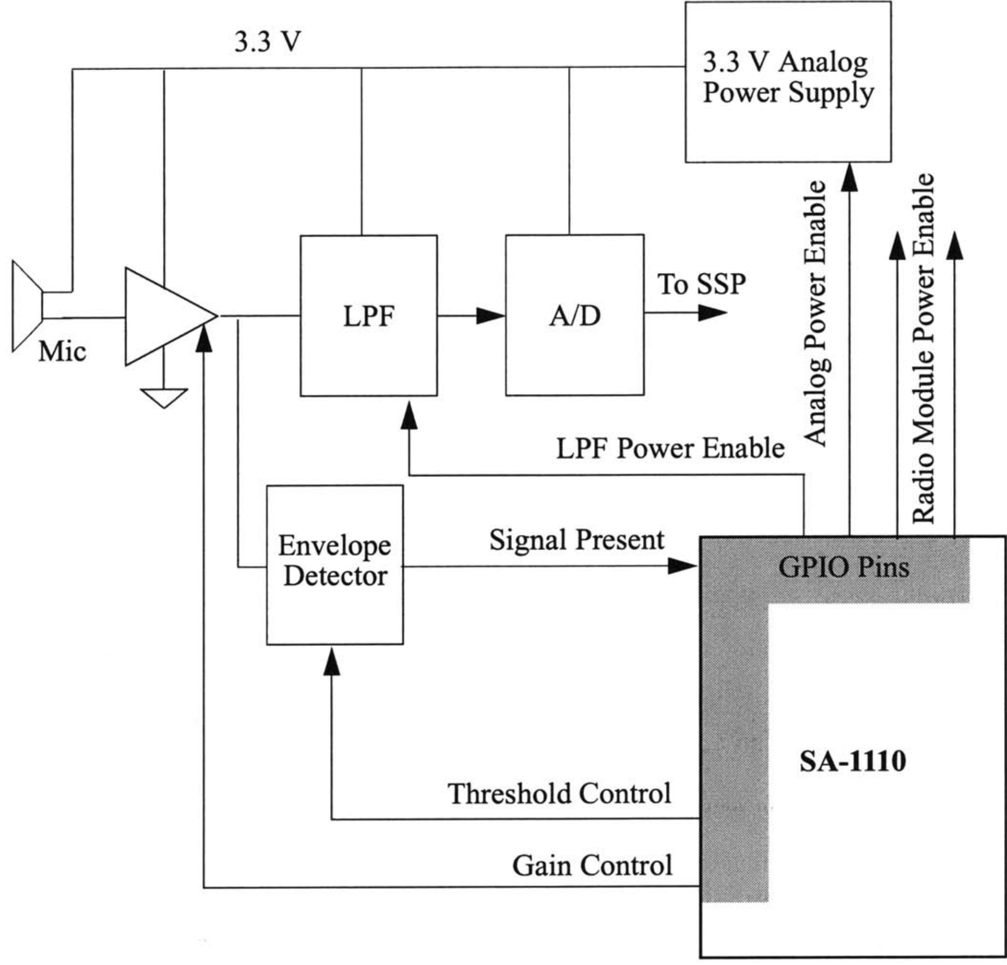


Figure 5-4: Idle power management hooks on the sensor processor board

5.2.3 Processor Power Modes

The SA-1110 contains power management logic that controls the transition between three different modes: run, idle and sleep. Each of these modes correspond to a reduced level of power consumption.

- *Run Mode* - This is the normal mode of operation for the SA-1110. All on-chip power supplies are on, all clocks are on and every on-chip resource is available. Under usual conditions the processor starts up in the run mode after a power-up or reset.
- *Idle Mode* - This mode allows an application to stop the CPU when not in use while continuing to monitor interrupt requests. The CPU clock is stopped and since the SA-

1110 is a fully static design, all state information in saved. When normal operation is resumed execution is started exactly where it was left. During idle mode all on-chip resources (real-time clock, OS timer, interrupt controller, GPIO, power manager, DMA and LCD controllers, etc.) are on. The PLL also remains in lock so that the processor can be brought in and out of the idle mode quickly.

- *Sleep Mode* - Sleep mode offers greatest power savings for the processor and consequently lowest functionality. When transitioning from run/idle to sleep mode the SA-1110 performs an orderly shutdown of on-chip activity, applies an internal reset to the processor, negates the power enable (PWR_EN) pin indicating to the external system that the power supply can be turned off. Running off the 32.768 KHz crystal oscillator, the sleep state machine watches for a pre-programmed wakeup event to occur. Sleep mode is entered in one of two ways - through software control or through a power supply fault. Entry into sleep mode is accomplished by setting the force sleep bit in the Power Manager Control Register (PMCR). This bit is set by software and cleared by hardware during sleep such that when the processor wakes up it finds the bit cleared. The entire sleep shutdown sequence takes about 90 μs.

Table 5-2 shows the power consumption in various modes of the SA-1110 processor at two different frequencies and the corresponding voltage specification as mentioned in [66]. Note that the minimum operating voltage required (as shown in Table 5-1) at the two frequencies is slightly lower than what is shown in Table 5-2. Idle mode results in about 75% power reduction while the sleep mode saves almost all the power.

Table 5-2: Power consumption in various modes of the SA-1110

Frequency	Supply Voltage (V)	Power Consumption Modes		
		Normal (mW)	Idle (mW)	Sleep (μA)
133	1.55	< 240	< 75	< 50
206	1.75	< 400	< 100	< 50

5.3 OS Architecture

The original eCos OS is designed to be completely scalable across platforms as well as within a given platform. Figure 5-5 shows the component architecture of eCos. Essentially, source level configuration allows the user to add or remove packages from a source repository based on system requirements. For example, the user might choose to remove math libraries and the resulting kernel will be leaner. The core eCos system consists of a number of different components such as the kernel, the C library, an infrastructure package, etc. Each of these provide a large number of configuration options, allowing application developers to build a system that matches the requirements of their particular application. To manage the potential complexity of multiple components and lots of configuration options, eCos has a component framework: a collection of tools specifically designed to support configuring multiple components. Furthermore, this component framework is extensible, allowing additional components to be added to the system at any time. The eCos Component Description Language (CDL) lets the configuration tools check for consistency in a given configuration and point out any dependencies that have not been satisfied.

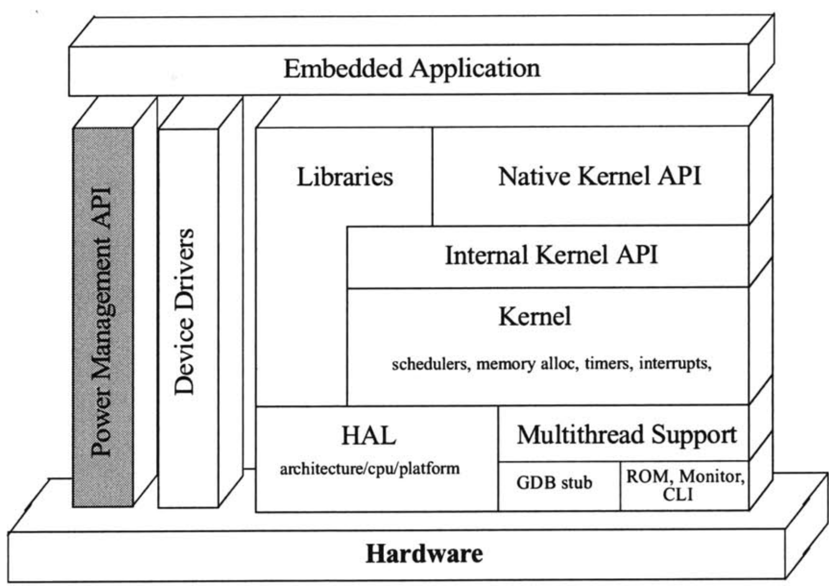


Figure 5-5: eCos architecture showing the power management layer that was added

5.3.1 Kernel Overview

At the core of the kernel is the scheduler. This defines the way in which threads are run, and provides the mechanisms by which they may synchronize. It also controls the means by which interrupts affect thread execution. To allow threads to cooperate and compete for resources, it is necessary to provide mechanisms for synchronization and communication. The classic synchronization mechanisms are mutexes/condition variables and semaphores [68]. These are provided in the eCos kernel, together with other synchronization/communication mechanisms that are common in real-time systems, such as event flags and message queues.

One of the problems that must be dealt with in any real-time systems is priority inversion. This occurs when a high priority thread is wrongly prevented from continuing by one at a lower priority. The normal example is of a high priority thread waiting at a mutex already held by a low priority thread. If the low priority thread is preempted by a medium priority thread then priority inversion has occurred since the high priority thread is prevented from continuing by an unrelated thread of lower priority. This problem got much attention recently when the Mars Pathfinder mission had to reset the computers on the ground exploration robot repeatedly because a priority inversion problem would cause it to hang. The eCos scheduler fixes this problem using a priority inheritance protocol. Here, the priority of the thread that owns the mutex is boosted to equal that of the highest priority thread that is waiting for it. This technique does not require prior knowledge of the priorities of the threads that are going to use the mutex, and the priority of the owning thread is only boosted when a higher priority thread is waiting. This reduces the effect on the scheduling of other threads, and is more optimistic than a priority ceiling protocol (where all threads that acquire the mutex have their priority boosted to some predetermined value). A disadvantage of this mechanism is that the cost of each synchronization call is increased since the inheritance protocol must be obeyed each time.

The kernel also provides exception handling. An exception is a synchronous event caused by the execution of a thread. These include both the machine exceptions raised by hardware (such as divide-by-zero, memory fault and illegal instruction) and machine exceptions raised by software (such as deadline overrun). The simplest, and most flexible,

mechanism for exception handling is to call a function. This function needs context in which to work, so access to some working data is required. The function may also need to be handed some data about the exception raised: at least the exception number and some optional parameters. As opposed to exceptions, which are synchronous in nature, interrupts are asynchronous events caused by external devices. They may occur at any time and are not associated in any way with the thread that is currently running and are harder to deal with. The ways in which interrupt vectors are named, how interrupts are delivered to the software and how interrupts are masked are all highly hardware specific. On the SA-1110, two kinds of interrupts are supported - FIQ (fast interrupts) and IRQ (regular interrupts). Both these interrupts can be masked.

Finally the kernel also provides a rich set of timing utilities such as counter, clocks, alarms and timers. The counter objects provided by the kernel provide an abstraction of the clock facility that is generally provided. Application code can associate alarms with counters, where an alarm is identified by the number of ticks until it triggers, the action to be taken on triggering, and whether or not the alarm should be repeated. Clocks are counters which are associated with a stream of ticks that represent time periods. Clocks have a resolution associated with them, whereas counters do not.

5.3.2 Application Programming Interface

Programs written for the sensor node do not have to satisfy any unusual requirements, but there are always some differences between a program written for a real-time operating system as opposed to one written for a time sharing, virtual memory system like UNIX or Windows NT. Any program which uses eCos system calls and µAMPS specific functions must include the following C header file:

```
#include <cyg/kernel/kapi.h> //Kernel APIs
#include <cyg/hal/uamps.h> // uAMPS API
```

and the programmer must make sure that the headers are available in the compiler's include path. The entry point for eCos user programs is usually `cyg_user_start()` instead of `main()`, although `main()` can be used if the ISO C library package is included. A summary of the native kernel functions available through the C language API

is shown in Table 5-3. Details about semantics and datatypes used in the functions are available in [69].

Table 5-3: Native kernel C language API

Function Type	Functions Available
Thread	cyg_scheduler_start(), cyg_scheduler_lock(), cyg_scheduler_unlock(), cyg_thread_create(), cyg_thread_exit(), cyg_thread_suspend(), cyg_thread_resume(), cyg_thread_yield(), cyg_thread_kill(), cyg_thread_delete(), cyg_thread_self(), cyg_thread_release(), cyg_thread_new_data_index(), cyg_thread_free_data_index(), cyg_thread_get_data(), cyg_thread_get_data_ptr(), cyg_thread_set_data(), cyg_thread_set_priority(), cyg_thread_get_priority(), cyg_thread_delay()
Exceptions	cyg_exception_set_handler(), cyg_exception_call_handler()
Interrupts	cyg_interrupt_create(), cyg_excepticyg_interrupt_attach(), cyg_interrupt_detach(), cyg_interrupt_get_vsr(), cyg_interrupt_set_vsr(), cyg_interrupt_disable(), cyg_interrupt_enable(), cyg_interrupt_mask(), cyg_interrupt_unmask(), cyg_interrupt_acknowledge(), cyg_interrupt_configure()
Counter Clocks Alarms	cyg_counter_create(), cyg_counter_delete(), cyg_counter_current_value(), cyg_counter_set_value(), cyg_counter_tick(), cyg_clock_create(), cyg_clock_delete(), cyg_clock_to_counter(), cyg_clock_set_resolution(), cyg_clock_get_resolution(), cyg_real_time_clock(), cyg_current_time(), cyg_alarm_create(), cyg_alarm_delete(), cyg_alarm_initialize(), cyg_alarm_enable(), cyg_alarm_disable()
Semaphores	cyg_semaphore_init(), cyg_semaphore_destroy(), cyg_semaphore_wait(), cyg_semaphore_trywait(), cyg_semaphore_timed_wait(), cyg_semaphore_post(), cyg_semaphore_peek()
Mutex	cyg_mutex_init(), cyg_mutex_destroy(), cyg_mutex_lock(), cyg_mutex_unlock(), cyg_mutex_release()
Memory Pools	cyg_mempool_var_create(), cyg_mempool_var_delete(), cyg_mempool_var_alloc(), cyg_mempool_var_try_alloc(), cyg_mempool_var_free(), cyg_mempool_var_waiting(), cyg_mempool_var_get_info(), cyg_mempool_fix_create((), cyg_mempool_fix_delete(), cyg_mempool_fix_alloc(), cyg_mempool_fix_timed_alloc(), cyg_mempool_fix_try_alloc(), cyg_mempool_fix_free(), cyg_mempool_fix_waiting(), cyg_mempool_fix_get_info()

Table 5-3: Native kernel C language API

Function Type	Functions Available
Message Boxes	cyg_mbox_create(), cyg_mbox_delete(), cyg_mbox_get(), cyg_mbox_timed_get(), cyg_mbox_tryget(), cyg_mbox_peek_item(), cyg_mbox_put(), cyg_mbox_timed_put(), cyg_mbox_tryput(), cyg_mbox_peek(), cyg_mbox_waiting_to_get(), cyg_mbox_waiting_to_put()
Flags	cyg_flag_init(), cyg_flag_destroy(), cyg_flag_setbits(), cyg_flag_maskbits(), cyg_flag_wait(), cyg_flag_timed_wait(), cyg_flag_poll(), cyg_flag_peek(), cyg_flag_waiting()

Table 5-4 shows most of the functions in the power management API. The functions are available to the application developer to enhance the power efficiency of his code. Syntax details and examples of benchmarks using the native kernel and power management API calls are available at [72].

Table 5-4: Primitive power management functions

Function Type	Functions Available
DVS	UAMPS_ENABLE_DVS(), UAMPS_SET_VOLTAGE(), UAMPS_SET_PROC_CLOCK(), UAMPS_CHECK_VCORE_STABLE(), UAMPS_SET_PROC_CLOCK(), uamps_set_proc_rate(), uamps_dvs_scheduler()
Shutdown	UAMPS_PERIPHERAL_POWER_ON(), UAMPS_PERIPHERAL_POWER_OFF(), UAMPS_V3_STANDBY_ON(), UAMPS_V3_STANDBY_OFF(), SA11X0_PWR_MGR_WAKEUP_ENABLE, SA11X0_PWR_MGR_GENERAL_CONFIG, SA11X0_PWR_MGR_CONTROL, uamps_set_proc_idle(), uamps_set_proc_sleep(), uamps_goto_sleep_state()

5.3.3 Web Based Application Development Tool

The application development toolchain is shown in Figure 5-6. The eCos source tree along with the µAMPS platform specific code is parsed using the `ecosconfig` tool to generate a build tree. Any package customizations are made at this configuration step. Details of the configuration tool and various available options can be obtained in [70]. The build tree is then compiled using the `arm-elf-gcc` cross compiler. If no conflicts exist in the configuration, the build process generates the `libtarget.a` library with the selected components. This library does not contain any user libraries (they must be linked separately). The same `arm-elf-gcc` cross compiler generates the object code for the

user application. During the final link step, the OS library is linked with the application object to produce an executable image that runs on the sensor node.

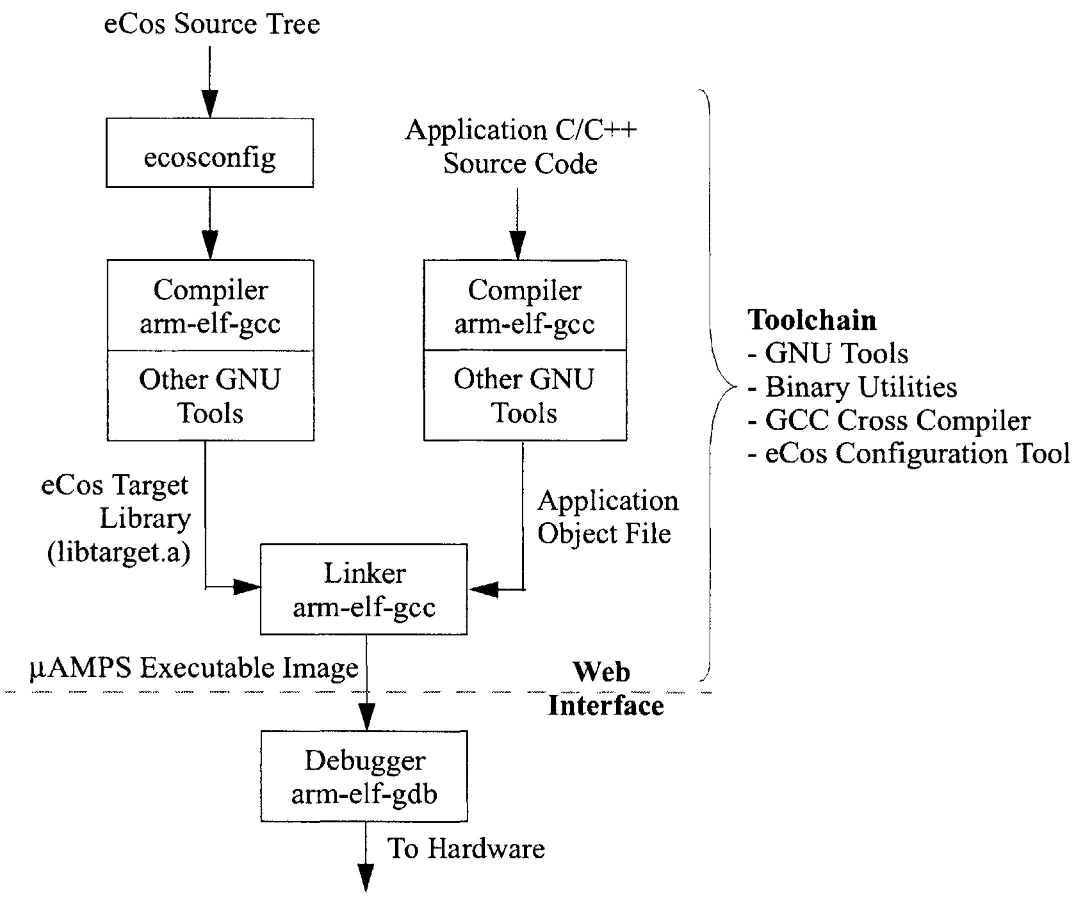


Figure 5-6: Application development toolchain for the μAMPS sensor node

To insulate the application developer from the having to install and maintain the configuration and compilation tools, and having to update the OS library with the latest version, as it evolves, we built a web interface to the entire tool chain. This interface allows a remote user to upload his application source code, and it produces the executable image linked with the OS and all appropriate libraries, which the user can easily download. In addition, the user can customize the OS based on the requirements of the application and optimize the memory requirement. A screenshot from the web based front end of the tool is shown in Appendix C.

5.4 System Level Power Management Results

5.4.1 Active Mode Power Savings

Figure 5-7 shows the power consumption of the sensor node in the fully active state (all modules on) as a function of the operating frequency of the SA-1110. The figure shows the power consumption using both DVS and just frequency scaling (which is done at a fixed core voltage of 1.65 V). The system power supply was 4.0 V. In the active mode, DVS is the primary source of power management. When running at the maximum operating voltage and operating frequency, the power consumption of the system is almost 1 W. *Active power management using DVS results in about 53% maximum system wide power savings*. The actual savings depends on the workload requirement.

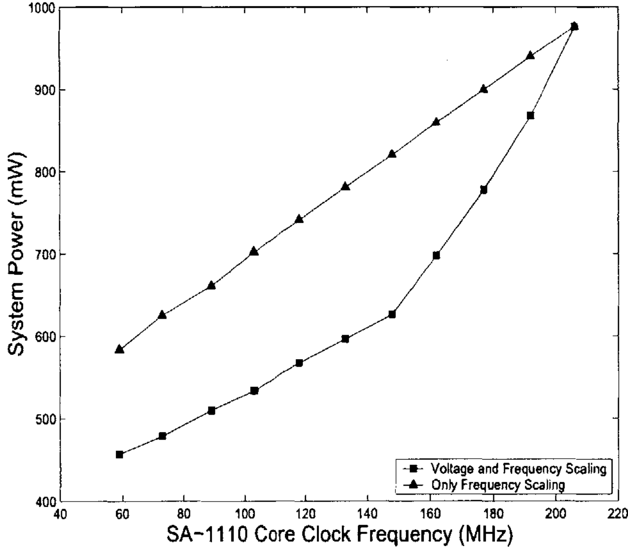


Figure 5-7: System level power savings from active power management using DVS

In Section 2.3 we introduced an important energy performance trade-off that was based on the fact that minimum energy consumption results when the processing rate variation is minimum. Minimum processing rate variation in turn implies a larger maximum performance hit. Figure 5-8 plots the relative battery life improvement as a function of the variance in workload. Each workload profile is Gaussian with a fixed average workload. The implications of our discussion in Section 2.3 is at once apparent. Although the aver-

age workload might be constant, the battery life improvement from DVS will degrade as the fluctuations in workload increase.

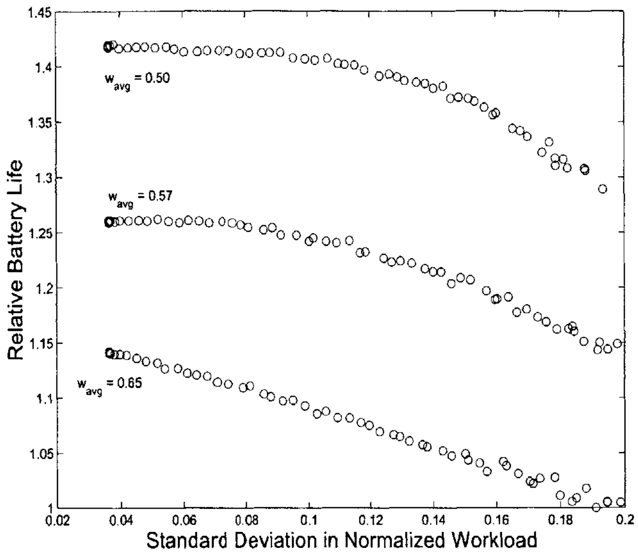


Figure 5-8: Degradation in DVS savings with increase in workload variance

5.4.2 Idle Mode Power Savings

Table 5-5: Measured power consumption in various modes of the sensor

	System Mode	Component Modes			Power (mW)
		Processor	Radio	Analog	
Active States	Active	Max Freq	on	on	975.6
	Low Active	Min Freq	on	on	457.2
Sleep States	Idle	idle	on	on	443.0
	Receive	idle	on	off	403.0
	Sense	idle	off	off	103.0
	Sleep	sleep	off	off	28.0

Table 5-5 shows the measured power consumption for the sensor node in various modes of operation. The sensor node can be classified as a processor power dominated architecture. The radio module follows the processor in power requirement (estimated at

about 70 mA at 3.3V). DVS can reduce system power consumption by 53%. Shutting down each of the components (analog power supply, radio module and the processor itself) results in another 44% power savings, i.e., *idle power management accounts for about 97% of system wide power savings*. The overall power savings attributed to various power management hooks have been shown in Figure 5-9.

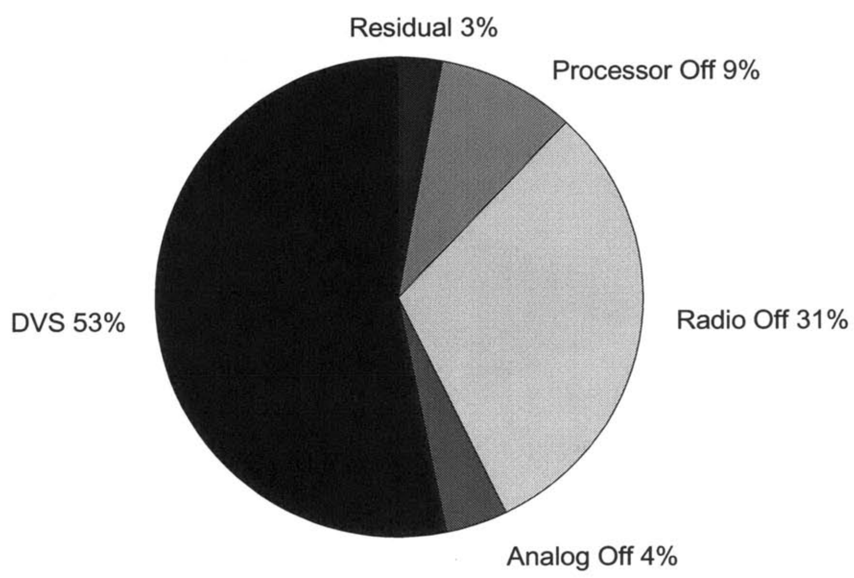


Figure 5-9: System level power savings distribution

The actual energy savings in the field depends significantly on processing rate requirements and event statistics. To estimate the energy savings from active mode power consumption, we would need an estimate of the workload variation on the system. (At present we do not have the required data from field trials). If we assume that the average workload requirement was 50%, with slow variations, the estimated energy savings is about 30%. Idle mode energy savings, on the other hand, can be significant. If we assume that the operational duty cycle is 1% the estimated energy savings is about $1-(0.01\times 0.7+0.99\times 0.03) = 0.9633 \approx 96\%$. This implies that sensor node battery life can be improved by a factor of over 27 (i.e., a node that lasts for a day with no power management will now last for almost a month)! With a 10% duty cycle, the battery life improvement is by a factor of about 10. The important point to observe is that the system is energy scalable, i.e., it has the right hooks to tune energy consumption based on compu-

tational load and sensing requirements. Figure 5-10 shows the factor by which battery life of the sensor node can be enhanced by using power management techniques as a function of the workload and duty cycle requirement.

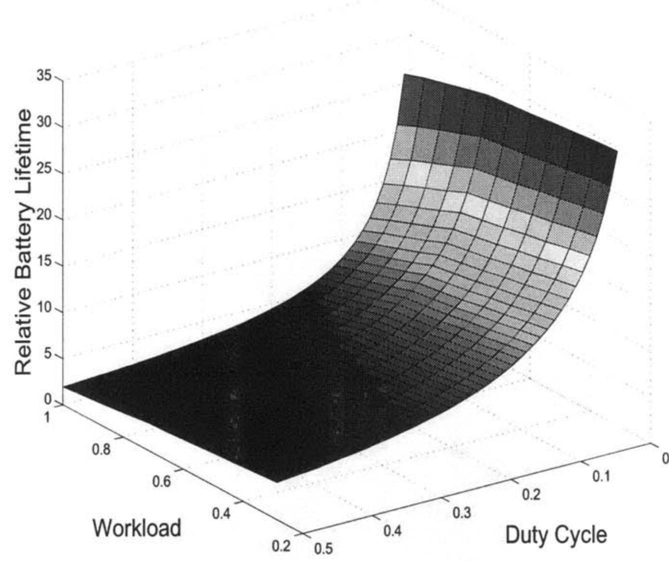


Figure 5-10: Battery life improvement in the sensor node compared to a node with no power management as a funtion of duty cycle and active workload

5.4.3 Energy Cost of Operating System Kernel Functions

The OS provides a rich API to the application for various tasks such as thread creation and manipulation, scheduling functions, mutex, mailbox and semaphore primitives and other functions to create and manipulate timers and alarms. A brief overview of these functions was presented in Section 5.3. We have estimated the energy cost of these kernel function calls. The average energy of each kernel function gives an estimate of the energy cost to the application developer. A detailed energy characterization of the kernel functions is included in Appendix A. Figure 5-11 shows the average, maximum and minimum system energy cost of various kernel functions grouped by category. The measurements have been made on a nominal system running at 59 MHz. The system power supply was the usual 4.0 V. Every function category consists of 10-20 different individual functions detailed in Appendix A. Each function was tested a number of times under different conditions. Although it is impossible to completely characterize a real-time operating system

(since latencies and therefore energy consumption depends heavily on interrupts, workloads, etc.), Figure 5-11 and Appendix A do give an estimate of the energy cost of various kernel API calls under nominal conditions. It can be seen that all kernel functions cost only a few μJoules of energy (corresponding to a few tens of μseconds execution time). Scheduling operations are the most efficient while thread manipulation and alarm functions are more expensive in comparison.

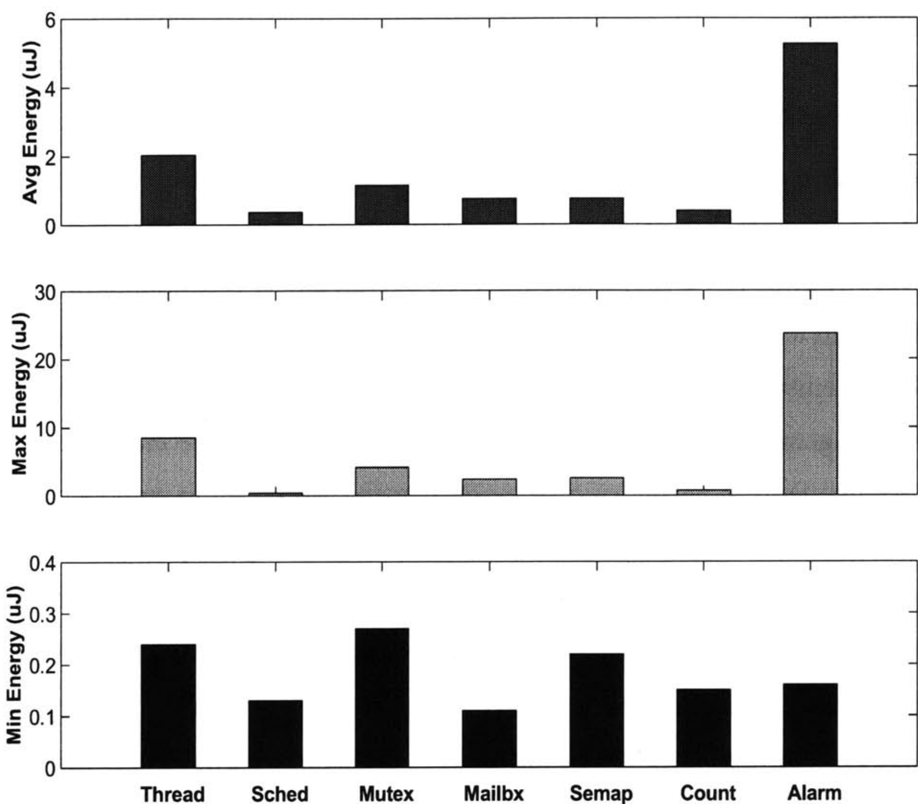


Figure 5-11: Average, maximum and minimum energy consumption of kernel function calls (grouped by category)

The energy cost of various OS calls can be used to estimate the overhead of the operating system. Consider a simple application with two threads. The first thread, T_A, implements radio and communication functions and executes every 100 ms. The second thread, T_B, implements the some signal processing algorithm on the sensor data and is invoked randomly with an average period of 5 ms. When no threads are runnable the processor executes a system idle thread. Therefore, over a one second interval, on an average, about

210 thread switches occur. The average energy overhead of thread scheduling can then be estimated as

$$Overhead = \frac{210 \times 1.79}{975.6 \times 1000} = 0.039\% \tag{5-1}$$

As can be seen, the overhead of OS calls is negligible. This is because most of these function take a few cycles to execute. On the other hand, having the OS greatly simplifies application development and the programmer does not have to worry about resource management issues.

5.5 Summary of Contributions

In this chapter we demonstrated the overall efficacy of system level software controlled power management using the μAMPS sensor node as an example. Although embedded systems differ significantly in their overall power consumption and power budgets allocated to various components, the sensor node is a generic example of an energy constrained embedded system with computation/communication capabilities. As such, the power management schemes proposed for the sensor node can be incorporated into other embedded systems to enable an 'energy-on-demand' approach to computation. We demonstrated that 50% system wide active mode energy savings is possible using DVS and another 40% energy savings results from shutting down components that are not required. Careful system design, with the right hardware hooks, can easily result in system energy scaling down by an order of magnitude in the idle mode. Given a low duty cycle of operation, this can significantly enhance the battery lifetime of an embedded system.

In terms of implementation, we demonstrated operating system controlled DVS on the StrongARM SA-1110 processor. The processor, as such, is not designed for run-time scaling of core voltage and clock frequency (usually the frequency is fixed at startup time). The Intel XScale [71] processors will perhaps support this feature. System level energy measurements in various modes were obtained to get an estimate of idle power savings. Energy characterization of kernel function calls was performed to quantify the cost of various real-time operating system tasks. A web based infrastructure has also been designed to enable users to develop applications using the power management features and pre-built scalable operating system kernels [72].

Chapter 6

Software Energy Measurement

In the previous chapters, we concentrated on the software control aspects of an energy constrained system. We developed techniques that the operating system could use to improve the energy efficiency of a system both in its active and idle states. In the next few chapters, we concentrate on the application software that runs on a system. We start out with a software energy estimation methodology that can be used to quantify the energy consumption of a piece of application code. Later we develop techniques to improve the energy efficiency of the application code.

In this chapter, we propose a software energy estimation framework based on actual energy measurements on the StrongARM SA-1100 and SH-4 processor boards. Software energy estimation is an integral component in any energy constrained embedded system and can be used to estimate the battery lifetime of the system. The estimation framework has been incorporated into a web-based tool, JouleTrack, that allows a user to upload a piece of C code and outputs energy measurements on the remote browser [73][74].

6.1 Factors Affecting Software Energy

Software energy estimation is the first step in the exploration of the design space in software energy optimization. Robust models that predict the energy cost of software are non-trivial to determine because of the dependence of energy consumption on the following parameters.

- *Target processor and Instruction Set Architecture (ISA)* - Software energy is an obvious function of the processor, data path and instruction set used. For example, the StrongARM microprocessor does not have a dedicated floating point unit. The energy required by a floating point operation on the StrongARM will be more than that on an equivalent processor with a well designed floating point unit. This is because a floating point operation on the StrongARM uses a software emulation to implement it and requires two orders of magnitude more time to execute than an

equivalent integer operation. Other architectural parameters which strongly affect the energy consumption of a processor are cache size, datapath width, number of functional units (multipliers, shifters), register file size, legacy support hardware, multimedia extension support, etc. In general, it is practically impossible to predict how much energy a piece of software will consume on another processor given the energy consumption profile on one processor, without some prior calibration and measurement on the other one.

- *Operating voltage and frequency* - As we have seen in previous chapters, the switching energy consumed depends linearly on the operating frequency and quadratically on supply voltage.
- *Leakage issues and duty cycle* - As the threshold voltages scale, leakage energy will become more dominant. Already, leakage accounts for 10% of energy dissipation on the StrongARM at 100% duty cycle. At 10% duty cycle the switching and leakage energy components become comparable (as we will show). Effective macro leakage models (module wise, if possible) would have to be developed that can isolate switching and leakage energy components before techniques can devised to minimize their effect without significant loss in performance.

6.2 Previous Work

Software energy estimation through exhaustive instruction energy profiling was first proposed in [75]. The basic experimental setup used in [75] is shown in Figure 6-1. The instantaneous current drawn by a CPU varies rapidly with time showing sharp spikes around clock edges. This is because at clock edges the processor's circuits switch (i.e., get charged or discharged) resulting is switching current consumption. Expensive hardware with a lots of measurement bandwidth and low noise is required to accurately track the CPU instantaneous current consumption. From a energy measurement perspective, however, we are interested in average current consumption. To a first order, battery lifetime depends on the average current and the amount of stored energy in the cell. Measuring average current is simpler and can be done using a current meter. The current meter averages the instantaneous current over an averaging window and the corresponding readings

are stable average values. The average current itself varies as the program executes. The basic approach proposed in [75] can be summarized as follows:

- Measure the current drawn by the processor as it repeatedly executes a certain instruction or sequences of instructions. This is done by putting the sequence in a loop and measuring current values. The measured values are the instruction base current cost.
- The program is broken up into basic blocks and the base cost of each instance of a block is obtained by adding the base cost of instructions in the block. These costs are provided in a base cost table. Obtain a run-time execution profile (instruction trace) for the program. Using this information the number of times the basic block is executed is determined and overall base cost is computed.
- The effect of circuit state (inter-instruction effects) is incorporated by analyzing pairs of instructions. A cache simulation is also performed to determine stalls and a cache penalty is also added to the final estimate.

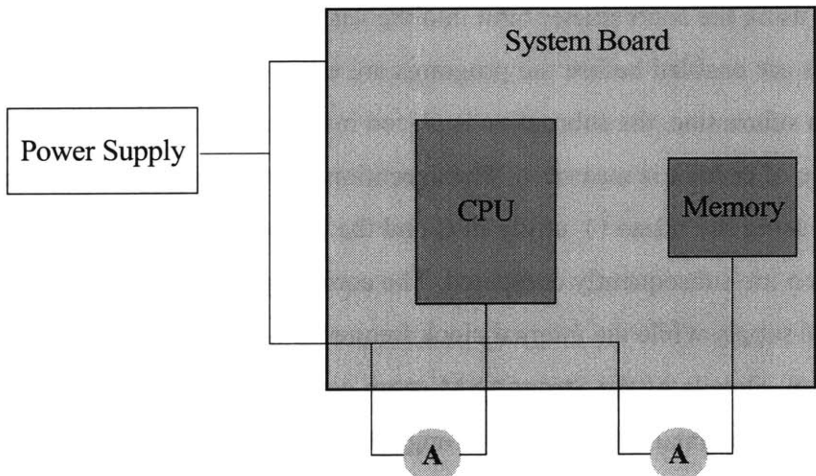


Figure 6-1: Experimental setup for instruction/program current measurement

The principal disadvantage of this approach is that it involves elaborate instruction trace analysis. Assuming an Instruction Set Architecture (ISA) with N instructions, N^2 instruction energy profiles have to be obtained to accurately determine base and inter-instruction costs. Moreover, most instructions have a lot of variations (e.g., based on

addressing modes, immediate operands, etc.) and exhaustive characterization is very time-consuming.

6.3 Proposed Methodology

6.3.1 Experimental Setup

Our basic experimental setup for estimation and verification purposes is similar to Figure 6-1. The StrongARM setup consists of the Brutus SA-1100 Design Verification Platform which is essentially the StrongARM SA-1100 microprocessor connected to a PC using a serial link. The SA-1100 consists of a 32-bit RISC processor core, with a 16 KB instruction cache and an 8 KB write-back data cache, a minicache, a write buffer, and a Memory Management Unit (MMU) combined in a single chip. It can operate from 59 MHz to 206 MHz, with a corresponding core supply voltage of 0.8 V to 1.5 V. Power supply to the StrongARM core was provided externally through a variable voltage sourcemeter. The I/O pads run at a fixed supply voltage. The ARM Project Manager (APM) is used to debug, compile and execute software on the StrongARM. Current measurements are performed using the sourcemeter built into the variable power supply. The instruction and data caches are enabled before the programs are executed. To measure the current that is drawn by a subroutine, the subroutine is placed inside a loop with multiple iterations till a stable value of current is measured. The execution time for multiple iterations is obtained accurately using the `time()` utility in C and the execution time per iteration and charge consumption are subsequently computed. The core supply voltage is altered directly from the external supply while the internal clock frequency of the processor is changed via software control. Details of the StrongARM setup can be found in [52][76][77]. Figure 6-2 shows the experimental setup for the StrongARM SA-1100 processor.

A similar board was setup for the Hitachi SH-4 processor using the SH7750 Solution Engine [78]. The SH7750 series (SH7750, SH7750S) is a 32-bit RISC microprocessor. It includes an 8 KB instruction cache, a 16 KB operand cache with a choice of copy-back or write-through mode, and an MMU with a 64-entry fully-associative unified Translation Lookaside Buffer (TLB). The SH7750 series has an on-chip bus state controller (BSC) that allows connection to DRAM and synchronous DRAM. Its 16-bit fixed-length instruction set enables program code size to be reduced by almost 50% compared with 32-bit

instructions. The SH-4 runs at 200 MHz fixed clock frequency at a core voltage of 1.95 V with 3.3 V for I/O pads. It is a 2-way superscalar CPU, i.e., it can execute 2 instructions per cycle. Unlike the SA-1100, the SH-4 features a fully integrated floating point unit with a separate bank of 16 single precision floating point registers. All this added performance comes at an increased energy cost. While the SA-1100 operates at 1 nJ/Instruction, the SH-4 requires 4.2 nJ/Instruction cycle.

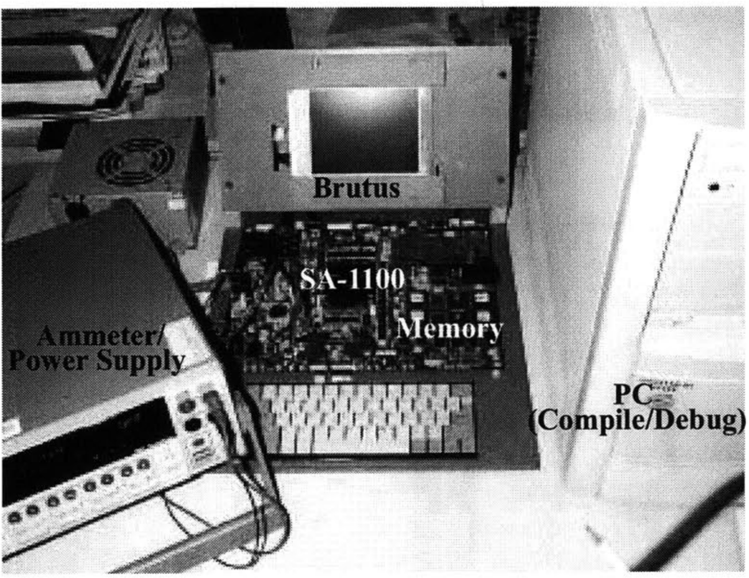


Figure 6-2: StrongARM SA-1100 experimental setup

6.3.2 Instruction Current Profiles

Our experiments on the StrongARM SA-1100 and Hitachi SH-4 microprocessors show that the variation in the current consumption across instructions is quite small. A lot of overheads are common across instructions and, as a result, the overall current consumption of a program is a weak function of the actual instruction stream and to a first order depends only on the operating frequency and voltage. Second order variations do exist but were measured to be less than 7% for a set of benchmark programs. Therefore, a complete instruction level trace analysis is unnecessary and a simple cycle accurate timing simulation can be used along with a simplified instruction/program current model to estimate overall energy consumption.

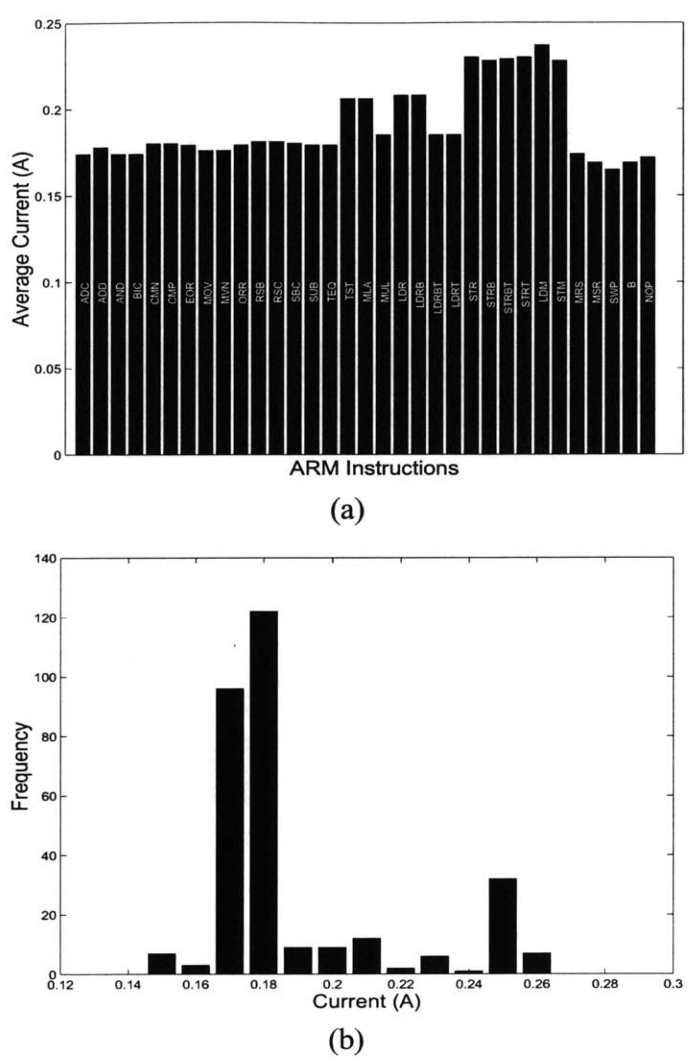


Figure 6-3: (a) Strong SA-1100 average instruction set current consumption
(b) Instruction set current distribution

Figure 6-3(a) shows the current consumption of all the instructions of the ARM instruction set on SA-1100. Each of the 33 current values are themselves the averages of the various addressing modes and inputs in which the instruction can be executed, accounting for a total of about 280 data points (the overall distribution is shown in Figure 6-3(b)). The important point to observe is that the current consumptions are pretty uniform. On an average, arithmetic and logical instructions consume 0.178 A, multiplies 0.196 A, loads 0.196 A, stores 0.229 A, while the other instructions consume about 0.170 A. The total variation in current consumption is 0.072 A, which is 38% of the overall aver-

age current consumption. Figure 6-3(b) shows the distribution of current consumption based on the 280 data points. All current measurements were done at 1.5 V core supply and 206 MHz clock frequency.

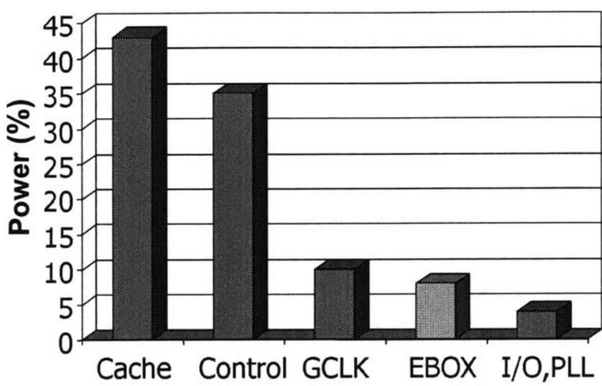


Figure 6-4: Distribution of power consumption in the ARM core [79]

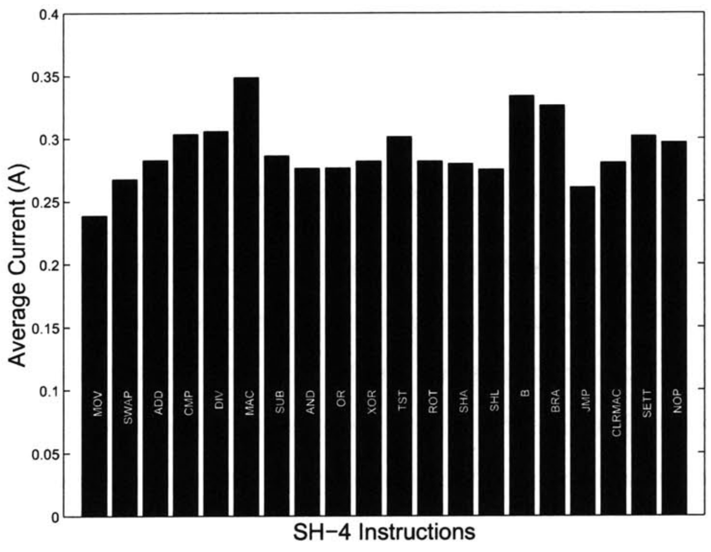


Figure 6-5: Hitachi SH-4 average instruction set current consumption

Although the small variation in instruction currents might appear surprising at first, a deeper analysis reveals that this ought to be expected. Figure 6-4 shows the distribution of power consumption across various blocks in the ARM core [79]. Between the cache, control, global clock (GCLK) and I/O circuits, almost 95% power is consumed. Since these

overheads are common to all instructions, the variation of current consumption between instructions is small. The execution box (EBOX) is where the instruction is implemented.

Figure 6-5 shows the core current consumption measured for the Hitachi SH-4 processor running at 2.0 V core power supply and 200 MHz clock frequency. The average instruction current is 0.29 A, with a variation of 0.11 A, which, once again, is 38% of the average. The small variation in instruction current can again be explained using a similar argument.

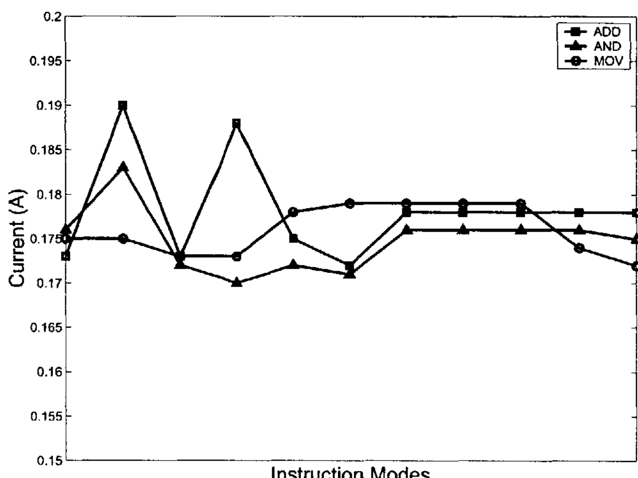


Figure 6-6: Current variations within instructions on the StrongARM

The current variation within an instruction (as a function of addressing modes and data) is even smaller. Figure 6-6 shows the current consumption of 3 different instructions as a function of various addressing modes and data on the SA-1100. In general, we observed that to a first order the instruction current consumptions are independent of the addressing modes or operands. We now propose a simple and fast technique to estimate software energy. Our experiments indicate an accuracy within 3% of actual measurements.

6.3.3 First Order Model

While the instruction level current consumption has a variation of about 38%, the variation of the current consumption between programs is much less. Figure 6-7 shows the current consumption of six different benchmark programs at different supply voltage and frequency levels on the StrongARM. The maximum current variation is about 8%. This

implies that, to a first order, current consumption of a piece of code is independent of the code and depends only on the operating voltage and frequency of the processor. The first order software energy estimation model is then simply

$$E_{tot} = V_{dd} I_0 (V_{dd}, f) \Delta t \tag{6-1}$$

where V_{dd} is the supply voltage and Δt is the program execution time.

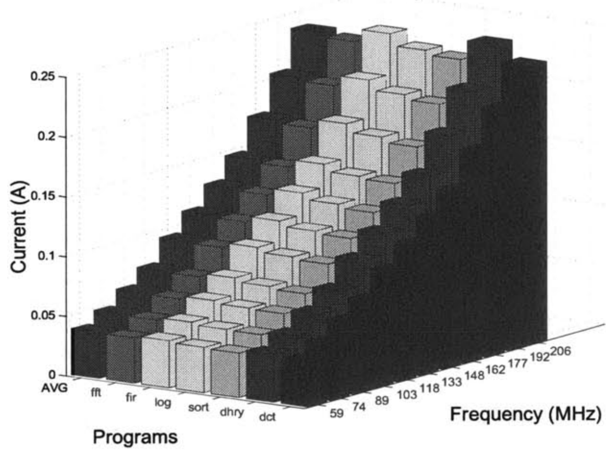


Figure 6-7: Program current consumption as a function of operating point

6.3.4 Second Order Model

While the current variation across programs is quite small in the StrongARM, it might be significant in datapath dominated processors. For example, the current consumption of the multiply instruction in DSPs will be far greater than the current consumption of other instructions. In such cases a simple model like Equation 6-1 will have significant error. The following second order model is proposed.

Let C_k ($k \in [0, K-1]$) denote K energy differentiated instruction classes in a processor. Energy differentiated instruction classes are partitions of the instruction set such that the average current consumption of any a class is significantly different from that of another class and the current variation within a class is small. Class partitions can also be done on the basis of different cycles (e.g., instruction, data access, etc.). Let c_k denote the fraction of total cycles in a given program that can be attribute to instructions/cycles in the class C_k, i.e., $\sum c_k = 1$. The second order current consumption is estimated as

$$I(V_{dd}, f) = I_0(V_{dd}, f) \sum_{k=0}^{K-1} w_k c_k \tag{6-2}$$

where w_k are a set of weights. Let $\boldsymbol{W}$ represent the vector $[w_0, w_1, .., w_{K-1}]$. Let P_n ($n \in [0, N-1]$) represent a set of N benchmark programs, $\boldsymbol{C}_n$ denote the cycle fractions vector for program P_n, i.e., $[c_0^n, c_1^n, \ldots, c_{K-1}^n]$ and I_n denote its average current consumption. Based on Equation 6-2, we can express the current vector $\boldsymbol{I}$ in the following form.

$$\boldsymbol{I} = I_0(V_{dd}, f)\boldsymbol{C}\boldsymbol{W} \tag{6-3}$$

where $\boldsymbol{I}$ is the average current $[I_0, I_1, \ldots, I_{N-1}]$ for the N programs, $\boldsymbol{C}$ is an $N \times K$ matrix with $\mathbf{C}_n$ as the nth row. The weights can be solved for in a least mean square sense using the pseudo-inverse

$$\boldsymbol{W} = \frac{1}{I_0(V_{dd}, f)}(\boldsymbol{C}^T\boldsymbol{C})^{-1}\boldsymbol{C}^T\boldsymbol{I} \tag{6-4}$$

If the instruction classes are a valid energy differentiated partition, the weighting vector $\boldsymbol{W}$ will reflect the energy differentiation. The maximum current prediction error will also go down considerably.

On the StrongARM SA-1100, we partitioned the cycles into 4 classes - (i) Instruction, (ii) Sequential memory access (accesses which are related to previous ones) (iii) Non sequential accesses, and, (iv) Internal cycles. Current measurements were done for six benchmark programs running at all possible frequency and voltage combinations. The weighting vector is shown in Table 6-1. The average current drawn at each operating frequency of the StrongARM is shown in Figure 6-8. The StrongARM operates at 11 discrete frequency levels. The minimum operating voltage required is also shown and is almost linear with frequency. In fact, the normalized operating voltage and frequency are related as

$$\bar{V}_{dd} = 0.66\bar{f} + 0.33 \tag{6-5}$$

where $\bar{V}_{dd}$ and $\bar{f}$ are the normalized to their respective maximum values.

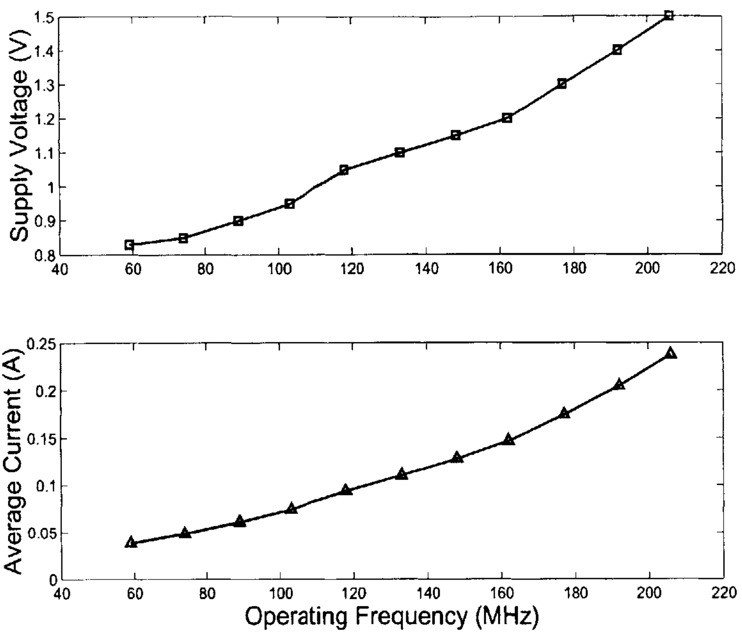


Figure 6-8: Average current and supply voltage at each operating frequency of the StrongARM SA-1100

The weighting factors can be interpreted as follows. For programs where instruction cycles and non-sequential memory accesses dominate, the current consumption is higher than the average current at that operating point. Internal cycles and sequential memory access dominated programs will have a lower than average current consumption. The current prediction error with a second order model can be reduced to less than 2%. The advantage is that this accuracy comes for free. No elaborate instruction level profiling is required. Such cycle counts as the ones shown in Table 6-1 are easily obtained using simulators/debuggers available with standard processors. Figure 6-9 shows the overall improvement of current prediction accuracy on 66 test points. It can be seen that the current prediction is better in every case (the maximum error of 4.7% using a first order model is reduced to 2.3%). The effectiveness of the current weighting scheme will become more pronounced in processors having a wider variation in average current consumption.

Table 6-1: Weighting factors for $K = 4$ on the StrongARM

Class	Weight	Value
Instruction	w_1	2.1739
Sequential memory access	w_2	0.0311
Non-sequential memory access	w_3	1.2366
Internal cycles	w_4	0.8289

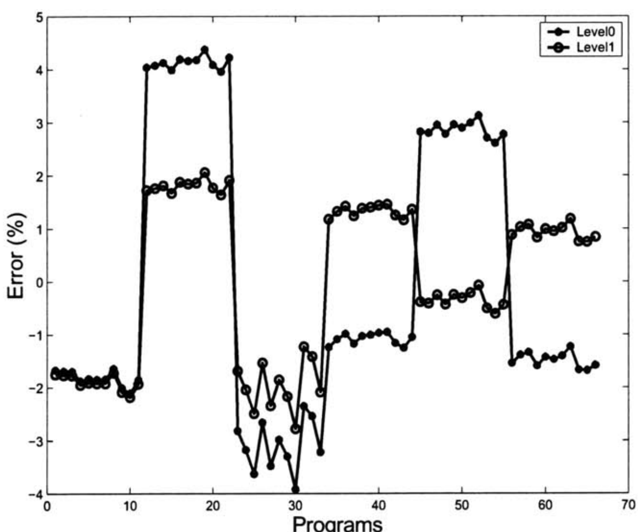


Figure 6-9: First and second order model prediction errors

6.4 Leakage Energy Measurement

6.4.1 Principle

The power consumption of a subroutine executing on a microprocessor can be macroscopically represented as

$$P_{tot} = P_{dyn} + P_{stat} = C_L V_{dd}^2 f + V_{dd} I_{leak} \tag{6-6}$$

where P_{tot} is the total power, which is the sum of the static and dynamic components, C_L is the total average capacitance being switched by the executing program per clock cycle, and f is the operating frequency (assuming that there are no static bias currents in the

microprocessor core) [9]. Let us assume that a subroutine takes Δt time to execute. This implies that the energy consumed by a single execution of the subroutine is

$$E_{tot} = P_{tot}\Delta t = C_{tot}V_{dd}^2 + V_{dd}I_{leak}\Delta t \tag{6-7}$$

where C_{tot} is the total capacitance switched by executing subroutine. Clearly, if the execution time of the subroutine is changed (by changing the clock frequency), the total switched capacitance, C_{tot}, remains the same. Essentially, the integrated circuit goes through the same set of transitions except that they occur at a slower rate. Therefore, if we execute the same subroutine at different frequencies, but at the same voltage, and measure the energy consumption we should observe a linear increase as the execution time increased, with the slope being proportional to the amount of leakage.

6.4.2 Observations

The subroutine chosen for execution was the decimation-in-time Fast Fourier Transform (FFT) algorithm because it is a standard, computationally intensive, DSP operation. The microprocessor, therefore, runs at maximum 'horsepower'. The execution time for a $N = 1024$ point FFT on the StrongARM is a few tenths of a second and scales as $O(N\log N)$. To obtain accurate execution time and stable current readings, the FFT routine was run a few hundred times for each observation. A total of eighty different data points corresponding to different supply voltages between 0.8 V and 1.5 V and operating frequencies between 59 MHz and 206 MHz were compiled.

Figure 6-10 illustrates the implications of Equation 6-7. When the operating frequency is fixed and the supply voltage is scaled, the energy scales almost quadratically. On the other hand, when the supply voltage is fixed and the frequency is varied the energy consumption decreases linearly with frequency (i.e., increases linearly with the execution time) as predicted by Equation 6-7. Not all frequency and voltage combinations are possible. For example the maximum frequency of the StrongARM is 206 MHz and it requires a minimum operating voltage of 1.5 V. The line across the surface plot demarcates the possible operating regions from the extrapolated ones (i.e., the minimum operating voltage for a given frequency).

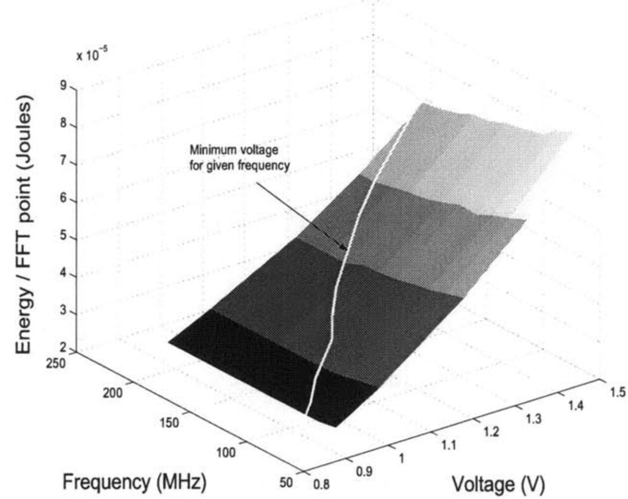


Figure 6-10: FFT energy consumption

We can measure the leakage current from the slope of the energy characteristics, for constant voltage operation. One way to look at the energy consumption is to measure the amount of charge that flows across a given potential. The charge attributed to the switched capacitance should be independent of the execution time, for a given operating voltage, while the leakage charge should increase linearly with the execution time. Figure 6-11 shows the measured charge flow as a function of the execution time for a 1024 point FFT. The amount of charge flow is simply the product of the execution time and current drawn. As expected, the total charge consumption increases almost linearly with execution time and the slope of the curve, at a given voltage, directly gives the leakage current at that voltage.

The dotted lines are the linear fits to the experimental data in the minimum mean-square error sense. At this point it is worthwhile to mention that the term "leakage current" has been used in an approximate sense. Truly speaking, what we are measuring is the total static current in the processor, which is the sum of leakage and bias currents. However, in the SA-1100 core, the bias currents are small and most of the static currents can be attributed to leakage. This assertion is further supported by the fact that the static current we measure has an exponential behavior as shown in the next section.

From the BSIM2 MOS transistor model [80], the sub-threshold current in a MOSFET is given by

$$I_{sub} = Ae^{\frac{(V_G - V_S - V_{TH0} - \gamma' V_S + \eta V_{DS})}{n' V_T}}\left(1 - e^{-\frac{V_{DS}}{V_T}}\right) \tag{6-8}$$

where

$$A = \mu_0 C_{ox} \frac{W_{eff}}{L_{eff}} V_T^2 e^{1.8} \tag{6-9}$$

and V_T is the thermal voltage, V_{TH0} is the zero bias threshold voltage, γ' is the linearized body effect coefficient, η is the Drain Induced Barrier Lowering (DIBL) coefficient and V_G, V_S and V_{DS} are the usual gate, source and drain-source voltages respectively. The important point to observe is that the subthreshold leakage current scales exponentially with the drain-source voltage.

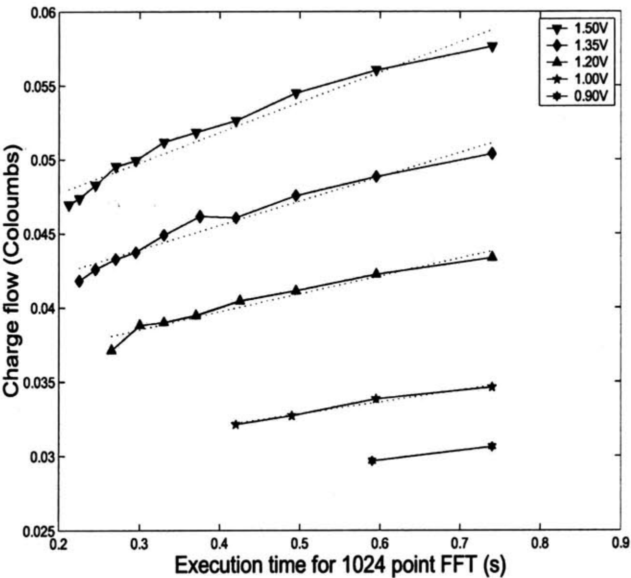


Figure 6-11: FFT charge consumption

The leakage current at different operating voltages was measured as described earlier, and is plotted in Figure 6-12. The overall microprocessor leakage current scales exponentially with the supply voltage. Based on these measurements the following model for the overall leakage current is proposed for the microprocessor core,

$$I_{leak} = I_0 e^{\frac{V_{dd}}{nV_T}} \quad (6\text{-}10)$$

where $I_0 = 1.196$ mA and $n = 21.26$ for the StrongARM SA-1100.

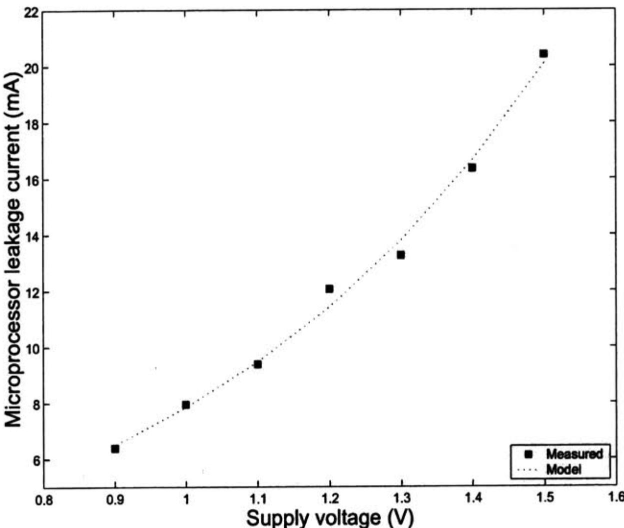


Figure 6-12: Leakage current variation

6.4.3 Explanation of Exponential Behavior

The exponential dependence of the leakage current on the supply voltage can be attributed to the DIBL effect. Consider the stack of NMOS devices shown in Figure 6-13. Equation 6-8 suggests that for a single transistor, the leakage current should scale exponentially with $V_{DS} = V_{DD}$ because of the DIBL effect. However since the ratio V_{DS} / V_T is larger than 2, the term inside the brackets of Equation 6-8 is almost 1. It has been shown in [81] that this approximation is also true for a stack of two transistors. With three or more transistors, the ratio V_{DS} / V_T for at least the lowest transistor becomes comparable to or even less than 1. Therefore, the term inside the bracket of Equation 6-8 cannot be neglected for such cases. The leakage current progressively decreases as the number of transistors in the stack increases and for a stack of more than three transistors the leakage current is small and can be neglected. It has further been shown in [81] that the ratio of the leakage currents for the three cases shown in Figure 6-13 can be written as

$$I_{l1} : I_{l2} : I_{l3} = 1.8e^{\frac{\eta V_{DD}}{nV_T}} : 1.8 : 1 \tag{6-11}$$

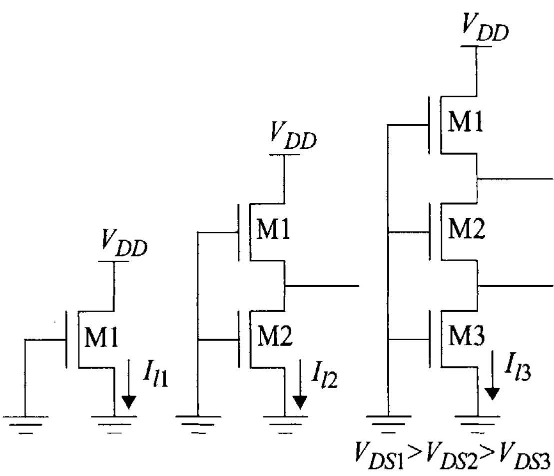


Figure 6-13: Effect of transistor stacking

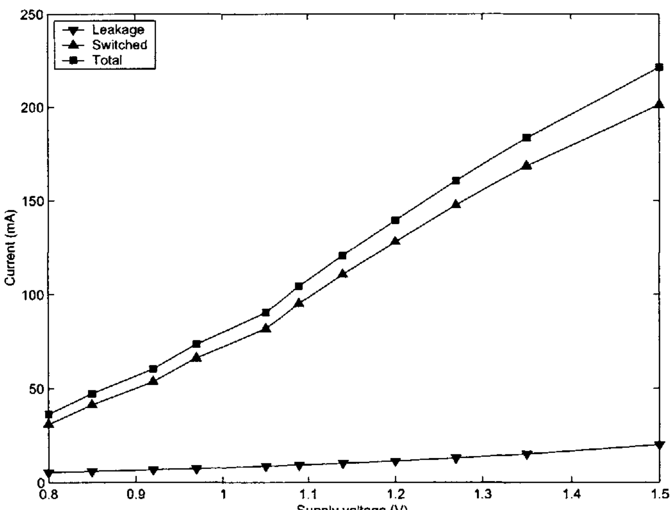


Figure 6-14: Static, dynamic and total current

Therefore, the leakage current of a MOS network can be expressed as a function a single MOS transistor (by accounting for the signal probabilities at various nodes and using the result of Equation 6-11). If the number of stacked devices is more than three, the leakage current contribution from that portion of the circuit is negligible. If there are three transistors stacked such that two of them are 'OFF' and one is 'ON' then the leakage anal-

ysis is the same as the stack of two 'OFF' transistors. For parallel transistors, the leakage current is simply the sum of individual transistor leakages. A similar argument holds for PMOS devices. Since, the leakage current of a single MOS transistor scales exponentially with V_{DD}, using the above arguments, we can conclude that the total microprocessor leakage current also scales exponentially with the supply voltage.

6.4.4 Separation of Current Components

Table 6-2: Leakage current measurements

V_{DD} (V)	I_{leak} (mA)		Error (%)
	Measured	Model	
1.50	20.41	20.10	1.50
1.40	16.35	16.65	-1.84
1.30	13.26	13.80	-4.04
1.20	12.07	11.43	5.27
1.10	9.39	9.47	-0.87
1.00	7.96	7.85	1.40
0.90	6.39	6.53	-1.70

Table 6-2 compares the measured leakage current with the values predicted by Equation 6-10. The maximum percentage error measured was less than 6% over the entire operating voltage range of the StrongARM, which suggests a fairly robust model. Based on the leakage model described by Equation 6-10, the static and dynamic components of the microprocessor current consumption can be separated as shown in Figure 6-14. The standby current of the StrongARM in the "idle" mode at 1.5 V is about 40 mA. This is not just the leakage current but also has the switching current due to the circuits that are still being clocked. On the other hand, this technique neatly separates the pure leakage component (assuming negligible static currents) from all other switching currents.For low threshold microprocessors like the StrongARM, it can seen that the leakage current is quite substantial (about 10% in this case). The leakage current, as a fraction of the total current, can be expressed as

$$\beta_{leak} = \frac{I_0 e^{\frac{V_{dd}}{nV_T}}}{I_0 e^{\frac{V_{dd}}{nV_T}} + kV_{dd}^{\alpha}} \tag{6-12}$$

where kV_{dd}^{α} models the switching current, has a very interesting profile. The leakage component, as a fraction of the total current, can be shown to have a minima at $V_{dd} = n\alpha V_T$, if we differentiate Equation 6-12 and solve for extrema. For the StrongARM, this is about 1.2 V as shown in Figure 6-15. The fact that such a minima occurs in measured current profiles further validates our model.

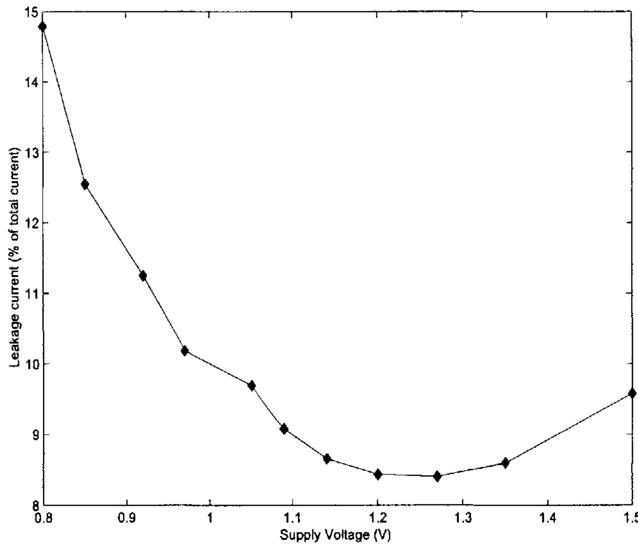


Figure 6-15: Leakage current fraction

6.4.5 Energy Trade-Offs

As the supply voltages and thresholds are reduced, system designers have to pay increasing attention to leakage currents. For the StrongARM, at maximum duty cycle and minimum voltage (for a given frequency), the leakage energy is about 10%. However, the leakage energy rises exponentially with supply voltage and decreases linearly with frequency as shown in Figure 6-16. Therefore, operating at a voltage, above the minimum possible, for a given frequency, is not advisable. In fact, if the operating voltage is suffi-

ciently high for the particular operating frequency, leakage energy can start to dominate switching energy as shown in Figure 6-16!

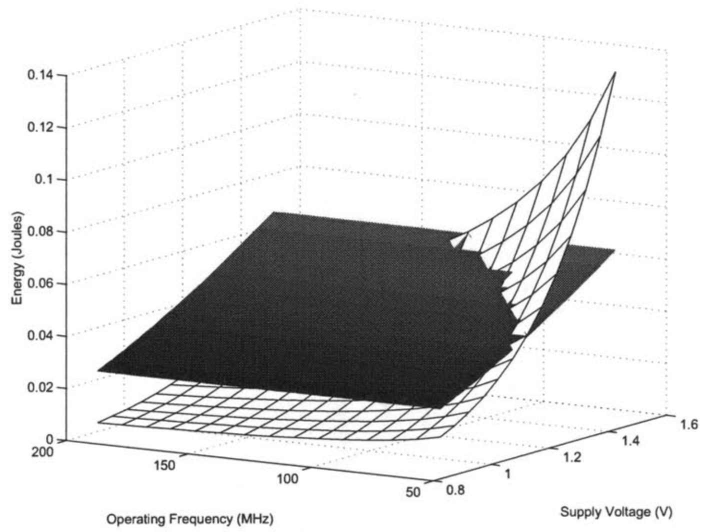


Figure 6-16: FFT energy components

For low duty-cycle systems, the overall energy consumption becomes increasingly dominated by leakage effects. The fixed task consumes a certain amount of switching energy per execution while the system leaks during the idle mode between tasks. Extensive clock gating techniques, such as those present in the StrongARM, reduce the unnecessary switching energy in the "idle" mode. The StrongARM also has a "sleep" mode where the supply voltage is reduced to zero for most circuits, and the processor state is stored. This significantly reduces the leakage problem. However, reverting to sleep mode between duty cycles may incur a lot of overhead (in terms of cycles and energy) or may not be supported by the target processor. Figure 6-17 illustrates the effect of duty cycle on the energy consumption of a system.

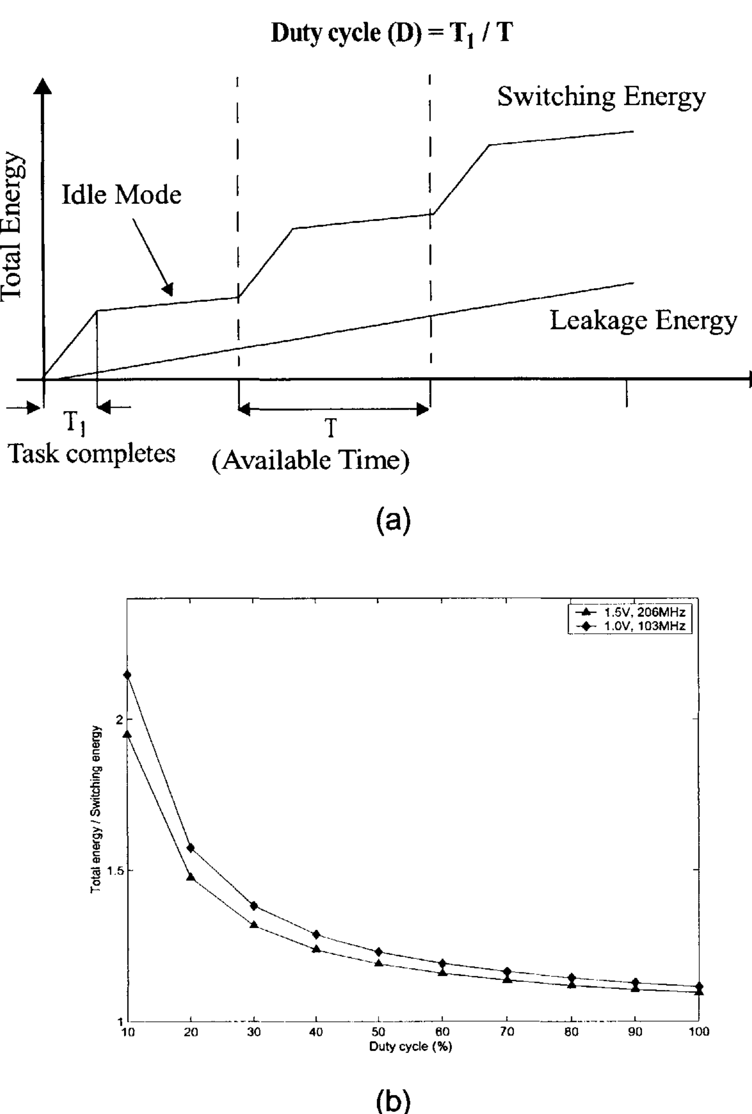


Figure 6-17: Low duty cycle effects

Suppose the system has to do an FFT every T seconds such that the execution time for the FFT is $T_1 \leq T$. After computing the FFT, the processor enters "idle" mode and switching activity is reduced by clock gating techniques. Leakage on the other hand is unaffected. Figure 6-17 (b) plots the ratio of total energy consumption to the switching energy, as a function of duty cycle. For a low duty cycle of 10% the ratio is about two for our FFT program, i.e., almost twice the amount of energy is used up for the same task compared to the 100% duty cycle case.

6.5 JouleTrack

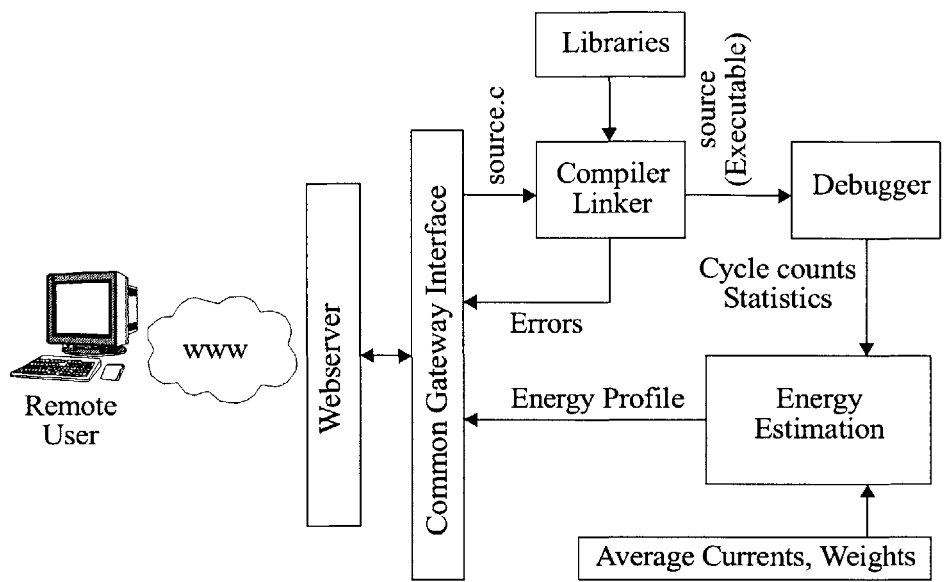


Figure 6-18: JouleTrack block diagram

The estimation techniques described in the previous sections were implemented in a web-based tool called JouleTrack. The tool is available at http://dry-martini.mit.edu/JouleTrack. The broad approach in the tool is summarized in Figure 6-18. The user uploads his C source code. The webserver uses Common Gateway Interface (CGI) scripts to create a temporary workarea for the user. His programs are compiled and linked with any standard C libraries. The user also specifies any command line arguments that the program might need along with a target operating frequency. Compiler optimization options are also available. The user can choose the current prediction model. Compile/link time errors are reported back by the CGI to the user. If no errors exist the program is executed on an ARM simulator which produces the program outputs (which the user can view), assembly listing (which can also be viewed) as well as run-time statistics like execution time, cycle counts, etc. These statistics are fed into an estimation engine which computes the energy profile and charts the various energy components. Important browser screenshots for the overall process are depicted in Appendix D.

Figure 6-19 shows the timing estimation engine used within Jouletrack. Energy is estimated as the product of execution time, average current and power supply voltage. The

average current is estimated using the methodology described in the previous sections. To measure the execution time, a cycle-accurate simulator for the processor core is used. A memory map specifies the memory latency and the addressable memory space of the system. If no memory map is specified, a default memory map (with 32 bit addressable memory space and uniform access time is used). The timing engine also requires the operating frequency for the processor. The functionality of the simulator core can be enhanced by adding various modules (e.g., profilers, tracers, co-processor and OS models) using a Remote Debug Interface (RDI).

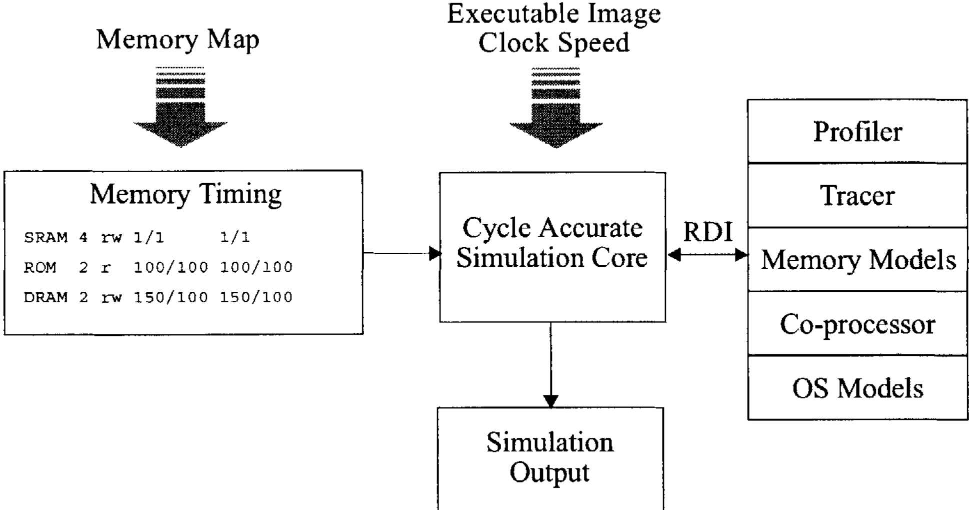


Figure 6-19: Timing estimation engine within Jouletrack

6.6 Summary of Contributions

Based on experiments done on the StrongARM and Hitachi processors we conclude that the common overheads present across instructions result in the variation in current consumption of different instructions being small. The variation in current consumption of programs is even smaller (less than 8% for the benchmark programs that we tried). Therefore, to a first order, we can assume that current consumption depends only on operating frequency and supply voltage and is independent of the executing program. A second order model that uses energy differentiated instruction classes/cycles is also proposed and it was shown that the resulting current prediction error was reduced to less than 2%. A

methodology for separating the leakage and switching energy components is also discussed. The proposed leakage current model has less than 6% error. Based on our experiments we conclude that as supply voltages and thresholds scale, leakage (static) components will become increasingly dominant. For the StrongARM, at 100% duty cycle, the leakage energy is about 10% and increases exponentially with supply voltage and decreases linearly with operating frequency. The proposed techniques were implemented in a web-based software energy estimation tool called JouleTrack.

Chapter 7

Energy Scalable Software

So far we have proposed operating system techniques to improve system energy efficiency and a methodology to quantify software energy consumption. From the discussion in Chapter 6, it is clear that since to a first order program current consumption is independent of what instructions are executing, traditional compiler optimizations cannot significantly improve the energy efficiency other than by reducing the execution time. Reducing execution time (i.e., improving performance) is what compiler optimizations do. It is fair to conclude that energy scaling is similar to time scaling in software. Therefore, rather than concentrating on compiler optimizations, we discuss algorithmic techniques and architectural hooks that can be exploited to enhance the energy efficiency of software.

It is highly desirable that we structure our algorithms and systems in such a fashion that computational accuracy can be traded off with energy requirement. At the heart of such transformations lies the concept of *incremental refinement* [82]. Consider the scenario where an individual is using his laptop for a full-motion video telephone application. Based on the current battery state, overall power consumption model and estimated duration of the call, the system should be able to predict its uptime [83]. If the battery life is insufficient, the user might choose to trade-off some quality/performance and extend the battery life of his laptop.

Similarly in the distributed sensor network scenario being used to monitor seismic activity from a remote basestation, it would be highly desirable to have energy scalable algorithms and protocols running on the sensor network. The remote basestation should have the capability to dynamically reduce energy consumption (to prolong mission lifetime if uninteresting events have occurred) by altering the throughput and computation accuracy. This type of behavior necessitates algorithmic restructuring so that every computational step leads us incrementally closer to the output.

In this chapter, we explore algorithmic techniques for efficient energy scalable computation [84]. A large class of algorithms, as they stand, do not render themselves to such

Energy-Quality (*E-Q*) scaling. Using simple modifications, the *E-Q* behavior of the algorithm can be modified such that if the available computational energy is reduced, the proportional hit in quality is minimal[1]. However, one must ensure that the energy overhead attributed to the transform is insignificant compared to the total energy consumption. It may be possible to do a significant amount of preprocessing such that the *E-Q* behavior is close to perfect, but we might end up with a situation where the overall energy consumption is higher compared to the unscalable system. This defeats the basic idea behind having a scalable system, viz., overall energy efficiency. We also briefly explore the power savings possible from exploiting parallelism in processors.

7.1 Energy Scalability Example

Consider the simple power series shown in Equation 7-1. Such power series are frequently encountered in Taylor expansions used to evaluate transcendental functions.

$$y = f(x) = 1 + k_1 x + k_2 x^2 + \ldots + k_N x^N \qquad (7\text{-}1)$$

A standard implementation of the algorithm would have an *N*-step loop that would multiply the current value of the computed power of x with x and accumulate the result in y. Let us assume we have to compute $f(2)$ for $N = 100$. If the k_i's are similar, even after N-1 steps in the loop, the value accumulated in y would be approximately 50% off from the final value since $2^N/f(2) \approx 1/2$. In terms of *E-Q* performance, the algorithm does not do well. Assuming that the amount of energy required to evaluate $f(2)$ on a processor is E_{max}, and that each step dissipates the same amount of energy (ignoring inter-instruction effects, etc.), we have about 50% computational accuracy after dissipating $(1\text{-}1/N).E_{max}$ energy. However, if we had to evaluate f(0.5), the most significant terms would occur in the first few steps in the loop and the *E-Q* behavior would be better. Based on the above analysis, we can conclude that transforming the algorithm, as shown in Table 7-1, will result in the most significant computations occurring early in the loop, as a result of which the computational energy could be reduced, without taking a significant hit in accuracy.

1. Since energy and time scaling are very similar, one could use similar arguments and transformations for Time-Quality scaling.

Figure 7-1 shows the *E-Q* graphs for the original and modified power series algorithm. It captures the all the basic ideas.

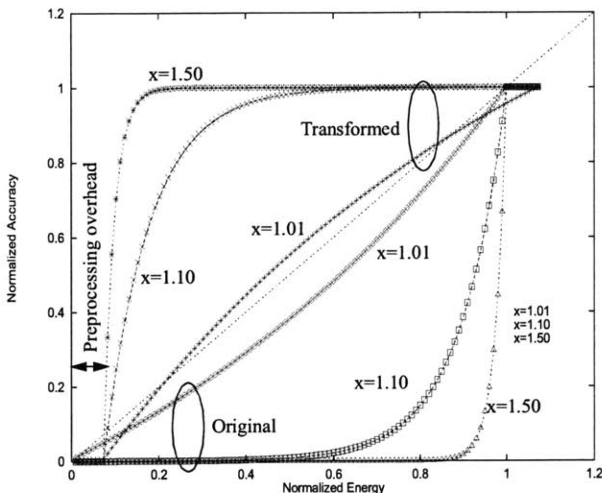


Figure 7-1: *E-Q* performance of power series algorithm

Table 7-1: Power series computation

Original Algorithm	Transformed Algorithm

xpowi = 1.0; y = 1.0;
for(i=1; i<N; i++) {
xpowi *= x;
y += xpowi*k[i];
}
	if(x>1.0) {
	xpowi = pow(x,N);
	y = k[N]*xpowi+1;
	for(i=N-1; i>0; i--) {
	xpowi /= x;
	y += xpowi*k[i]; }
	}
	else { // original algo
	}

- *Data Dependence* - *E-Q* behavior is, in general, data dependent. It is possible to come up with pathological cases where the transformed algorithm would have a *E-Q* behavior very close to the original. However, from an energy efficiency perspective, its the average *E-Q* performance that matters.
- *Concavity* - It is desirable to have an *E-Q* graph above the baseline ($E = Q$ on a normalized scale). This would imply that marginal returns in accuracy from successive units of computational energy is diminishing. Therefore, if the available energy is

reduced by 10%, the quality degradation is less that 10%, the lesser, the better.

- *Transformation Overhead* - There is an energy overhead associated with the transform which should be insignificant compared to the total energy

7.2 Formal Notions for Scalability

We now formalize the notion of a desirable *E-Q* behavior of a system. The *E-Q* graph of an algorithm is the function, *Q(E)*, representing some quality metric (e.g., mean-square error, peak signal-to-noise ratio, etc.) as a function of the computational energy $0 \le E \le E_{max}$. There may exist situations where the notion of a quality metric is unclear. However, in this paper, we are dealing with signal processing algorithms where the notion of a quality metric is usually unambiguous. Consider two algorithms (I and II) that perform the same function. Ideally, from an energy perspective, II would be a more efficient scalable algorithm compared to I if

$$Q_{II}(E) > Q_I(E) \quad \forall E \tag{7-2}$$

In most practical cases, Equation 7-2 will not hold over all energy values. As shown in Table 7-1, there might be a preprocessing overhead as a result of which the maximum energy consumptions might be different for the two cases (i.e., $E_{max,\ II} > E_{max,\ I}$). Nevertheless, as long as the Equation 7-2 holds over a significant range of computational energies, overall efficiency is assured.

Let us assume that there exists a quality distribution, $p_Q(x)$, i.e., from system statistics we are able to conclude that the probability that we would want a quality x is $p_Q(x)$. A typical quality distribution is shown in Figure 7-2. The average energy consumption per output sample can then be expressed as

$$\bar{E} = \int p_Q(x) E(x) dx \tag{7-3}$$

where *E(Q)* is the inverse of *Q(E)*. When the quality distribution is unknown, we would like the *E-Q* behavior to be maximally concave downwards (with respect to the energy axis), i.e.,

$$\frac{\partial^2 Q(E)}{\partial E^2} \le 0 \tag{7-4}$$

The E-Q behavior suggested by Equation 7-4 is not always attainable globally, i.e., across $0 \le E \le E_{max}$, as we will see subsequently. However, on an average case, for a given energy availability, E, we would like the obtainable quality $Q(E)$ to be as high as possible.

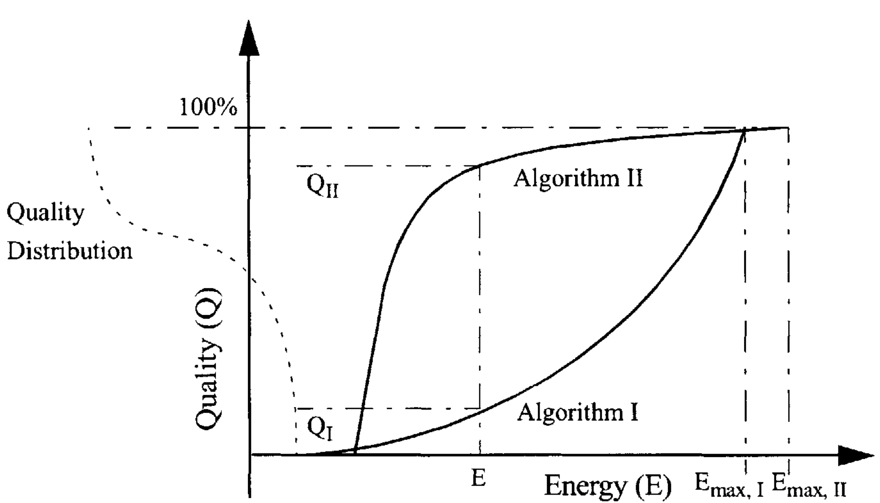


Figure 7-2: E-Q formal notions

7.3 Energy Scalable Transformations

We now demonstrate simple transformations that can significantly improve the E-Q behavior of an algorithm using three popular classes of signal processing applications - filtering, image decoding and beamforming.

7.3.1 Filtering Application

Finite Impulse Response (FIR) filtering is one of the most commonly used DSP operations. FIR filtering involves the inner product of two vectors one of which is fixed and known as the impulse response, $h[n]$, of the filter [85]. An N-tap FIR filter is defined by Equation 7-5.

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k] \tag{7-5}$$

Various low power and energy efficient implementations of the FIR filter have been proposed and implemented [86]. The approximate processing techniques proposed in [82]

reduce the total switched capacitance by dynamically varying the filter order based on signal statistics.

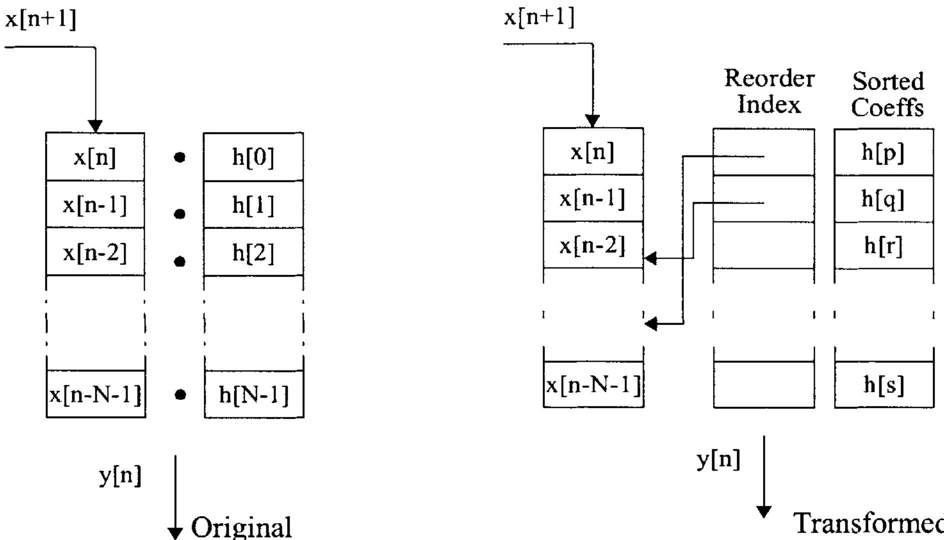


Figure 7-3: FIR filtering with coefficient reordering

When we analyze the FIR filtering operation from a pure inner product perspective, it simply involves N multiply and accumulate (MAC) cycles. For desired *E-Q* behavior, the MAC cycles that contribute most significantly to the output $y[n]$ should be done first. Each of the partial sums, $x[k]h[n-k]$, depends on the data sample and therefore its not apparent which ones should be accumulated first. Intuitively, the partial sums that are maximum in magnitude (and can therefore affect the final result significantly) should be accumulated first. Most FIR filter coefficients have a few coefficients that are large in magnitude and progressively reduce in amplitude. Therefore, a simple but effective *most-significant-first transform* involves sorting the impulse response in decreasing order of magnitude and reordering the MACs such that the partial sum corresponding to the largest coefficient is accumulated first as shown in Figure 7-3. Undoubtedly, the data sample multiplied to the coefficient might be so small as to mitigate the effect of the partial sum. Nevertheless, on an average, the coefficient reordering by magnitude yields a better *E-Q* performance than the original scheme.

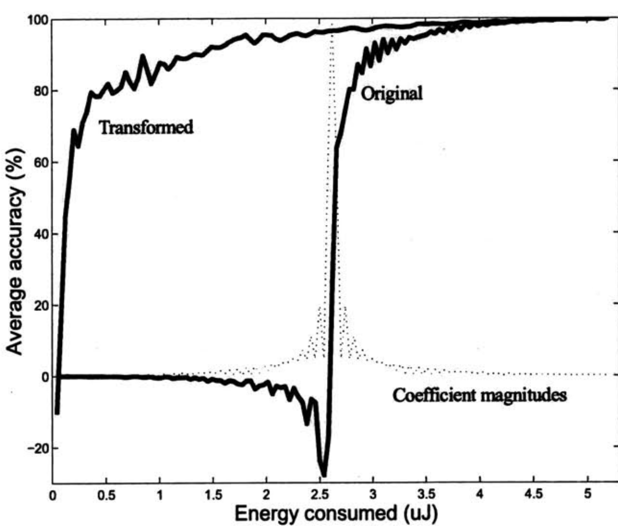


Figure 7-4: *E-Q* graph for original and transformed FIR filtering

Figure 7-4 illustrates the scalability results for a low pass filtering of speech data sampled at 10 KHz using a 128-tap FIR filter whose impulse response (magnitude) is also outlined. The average energy consumption per output sample (measured on the StrongARM SA-1100 operating at 1.5 V power supply and 206 MHz frequency) in the original scheme is 5.12 μJ. Since the initial coefficients are not the ones with most significant magnitudes the *E-Q* behavior is poor. Sorting the coefficients and using a level of indirection (in software that amounts to having an index array of the same size as the coefficient array), the *E-Q* behavior can be substantially improved. It can be seen that fluctuations in data can lead to deviations from the ideal behavior suggested by Equation 7-4, nonetheless overall concavity is still apparent. The energy overhead associated with using a level of indirection on the SA-1100 was only 0.21 μJ which is about 4% of the total energy consumption. Figure 7-5 shows the ratio of the energy consumed in the unsorted system to the sorted system for a given quality.

In FIR filtering, the input data samples are unknown *a priori*. The partial sum which is most significant is not completely deterministic until all of them have been computed. More sophisticated schemes could involve sorting both the data samples and the coefficients and using two levels of indirection to perform the correct inner product first by picking up the partial sum corresponding to the largest coefficient, then the one corre-

sponding to the largest data sample and so on. The overhead associated with such a scheme involves real time sorting of incoming samples. Assuming that we have a pre-sorted data array at time n, the next data sample $x[n+1]$ can be inserted into the right position using a binary search type technique which can be done in $O(logN)$. The scalability gains might not be substantial compared to the simpler scheme discussed before. However, in applications such as autocorrelation which involves an inner product of a data stream with a shifted version of itself, sorting both the vectors in the inner product would yield significant improvements in *E-Q* behavior.

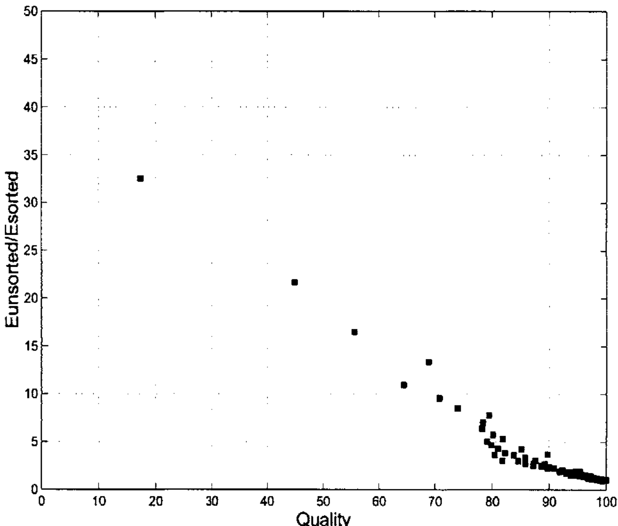


Figure 7-5: Energy inefficiency of unsorted system compared to the sorted case

7.3.2 Image Decoding Application

The Discrete Cosine Transform (DCT), which involves decomposing a set of image samples into a scaled set of discrete cosine basis functions, and the Inverse Discrete Cosine Transform (IDCT), which involves reconstructing the samples from the basis functions, are crucial steps in digital video [87]. The 64-point, 2-D DCT and IDCT (used on 8x8 pixel blocks in of an image) are defined respectively as

$$X[u, v] = \frac{c[u]c[v]}{4}\sum_{i=0}^{7}\sum_{j=0}^{7}x[i,j]\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \qquad (7\text{-}6)$$

$$x[i,j] = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} c[u]c[v]X[u,v]\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \quad (7\text{-}7)$$

DCT is able to capture the spatial redundancy present in an image and the coefficients obtained are quantized and compressed. Most existing algorithms attempt to minimize the number of arithmetic operations (multiplications and additions) usually relying on the symmetry properties of the cosine basis functions (similar to the FFT algorithm) and on matrix factorizations [88]. The *E-Q* behavior of these algorithms are not good as they have been designed such that computation takes a minimal yet constant number of operations. The Forward Mapping-IDCT (FM-IDCT) algorithm, proposed in [89], can be shown to have an *E-Q* performance that is much better than other algorithms. The algorithm is formulated as follows

$$\begin{bmatrix} x_{0,0} \\ x_{0,1} \\ \vdots \\ x_{8,8} \end{bmatrix} = X_{0,0}\begin{bmatrix} c_0^{0,0} \\ c_1^{0,0} \\ \vdots \\ c_{64}^{0,0} \end{bmatrix} + X_{0,1}\begin{bmatrix} c_0^{0,1} \\ c_1^{0,1} \\ \vdots \\ c_{64}^{0,1} \end{bmatrix} + \ldots + X_{8,8}\begin{bmatrix} c_0^{8,8} \\ c_0^{8,8} \\ \vdots \\ c_{64}^{8,8} \end{bmatrix} \quad (7\text{-}8)$$

where $x_{i,j}$ are the reconstructed pels, $X_{i,j}$ are the input DCT coefficients, and $[c_k^{i,j}]$ is the 64x64 constant reconstruction kernel. The improved *E-Q* behavior of the FM-IDCT algorithm can be attributed to the fact that most of the signal energy is concentrated in the DC coefficient ($X_{0,0}$) and, in general, in the low-frequency coefficients as shown in Figure 7-6. Instead of reconstructing each pixel by summing up all its frequency contributions, the algorithm incrementally accumulates the entire image based on spectral contributions from the low to high frequencies.

Figure 7-7 and Figure 7-8 illustrate the behavior of FM-IDCT algorithm. It is obvious from Figure 7-7 that almost 90% image quality can be obtained with as little as 25% of the total energy consumption. In terms of the overhead requirement, the only change that is required is that we now need to store the IDCT coefficients in a transposed fashion (i.e., all the low frequency components first, and so on).

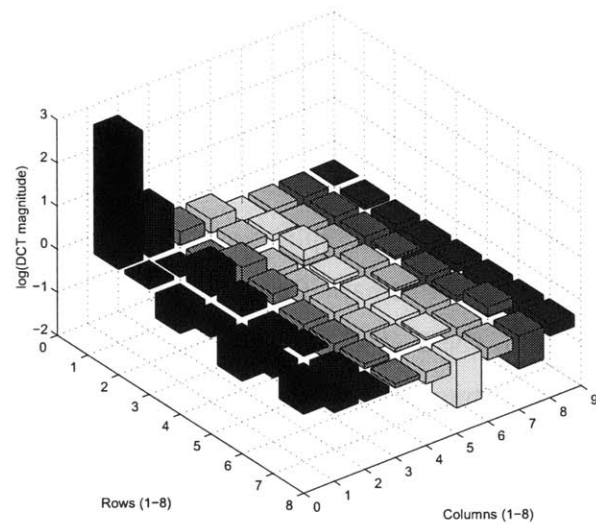


Figure 7-6: 8x8 DCT coefficient magnitudes averaged over a sample image

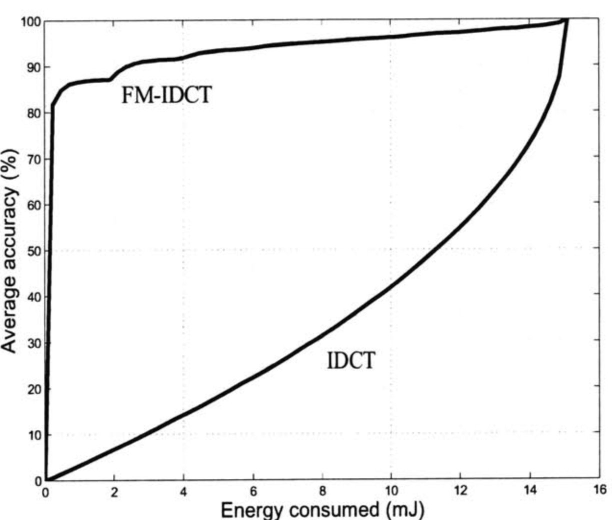


Figure 7-7: *E-Q* graph for FM-IDCT vs normal IDCT

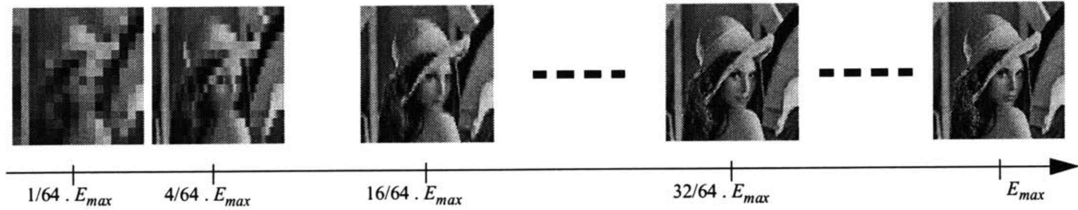


Figure 7-8: Illustrating the incremental refinement property with respect to computational energy of the FM-IDCT algorithm

7.3.3 Classification using Beamforming

Beamforming algorithms can be used to aggregate highly correlated data from multiple sensors into one representative signal. The advantages of beamforming is twofold. First, beamforming is used to enhance the desired signal while interference or uncorrelated sensor noise is reduced. This leads to an improvement in detection and classification of the target. Second, beamforming reduces redundant data through compression of multiple sensor data into one signal. Figure 7-9 shows a block diagram of a wireless network of *M* sensors utilizing beamforming for local data aggregation.

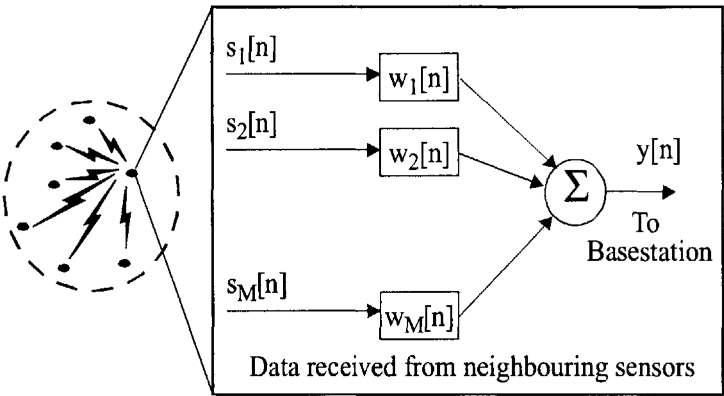


Figure 7-9: Beamforming for data aggregation

We have studied various beamforming algorithms that fall under the category of "blind beamforming" [90]. These beamformers provide suitable weighting functions, $w_i(n)$, to satisfy a given optimality criterion, without knowledge of the sensor locations. In this section we will show energy scalability for one particular blind beamforming algorithm, the Least Mean Squares (LMS) beamforming algorithm. The LMS algorithm uses a minimum mean squared error criterion to determine the appropriate array weighting filters. This algorithm is considered an optimum algorithm, and is highly suitable for power aware wireless sensor networks [91].

We will now show how algorithmic transformations can be used to improve the *E-Q* model for LMS beamforming[1]. Figure 7-10 shows our testbed of sensors for this example. We have an array of 6 sensors spaced at approximately 10 meters, a source at a distance of

1. The author would like to acknowledge Alice Wang of MIT for the beamforming experiment.

10 meters from the sensor cluster, and interference at a distance of 50 meters. We want to perform beamforming on the sensor data, measure the energy dissipated on the StrongARM SA-1100, calculate the matched filter output (quality), and provide a reliable model of the *E-Q* relationship as we vary the number of sensors in beamforming.

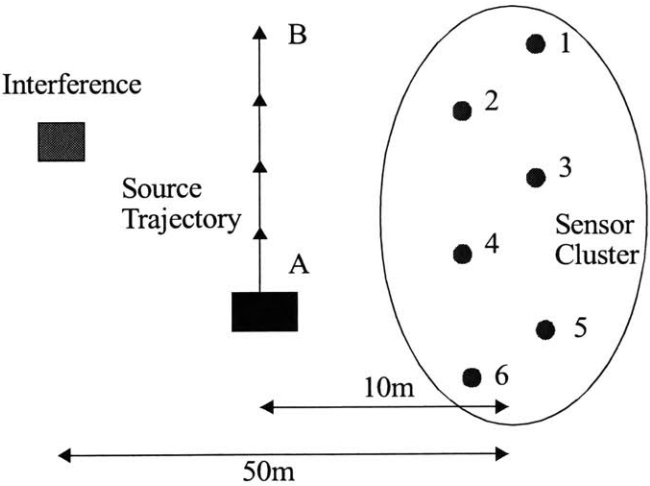


Figure 7-10: Sensor testbed

In Scenario 1, we will perform beamforming without any knowledge of the source location in relation to the sensors. Beamforming will be done in a pre-set order <1,2,3,4,5,6>. The parameter we will use to scale energy is n, the number of sensors in beamforming. As n is increased from 1 to 6, there is a proportional increase in energy. As the sensor moves from location A to B we take snapshots of the *E-Q* curve, shown in Figure 7-11. This curve shows that with a preset beamforming order, there can be vastly different *E-Q* curves, which leads to a poor Energy-Quality scalability. When the source is at location A, the beamforming quality is only at maximum when sensors 5 and 6 are beamformed. Conversely, when the source is at location B, the beamforming quality is close to maximum after beamforming two sensors. Therefore, for this setup, since the *E-Q* curve is highly data dependent, an accurate *E-Q* model for LMS beamforming is not possible.

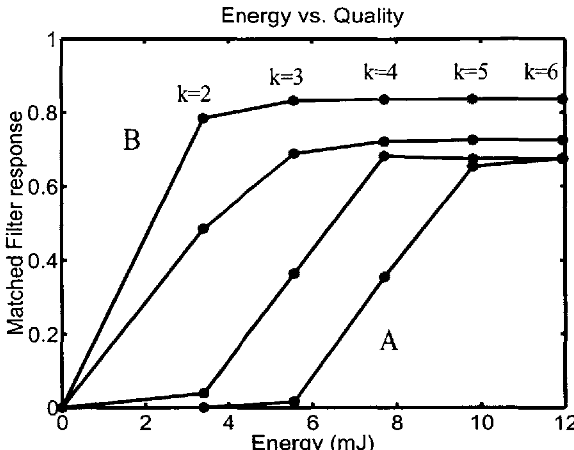


Figure 7-11: *E-Q* snapshot for Scenario 1

An intelligent alternative is to perform some initial pre-processing of the sensor data to determine the desired beamforming order for a given set of sensor data. Intuitively, we want to beamform the data from sensors which have higher signal energy to interference energy. Using the *most-significant-first transform*, which was proposed earlier, the *E-Q* scalability of the system can be improved. To find the desired beamforming order, first the sensor data energy is estimated. Then the sensor energies are sorted using a quicksort method. The quicksort output determines the desired beamforming order. Figure 7-12 shows a block diagram of the transformed system.

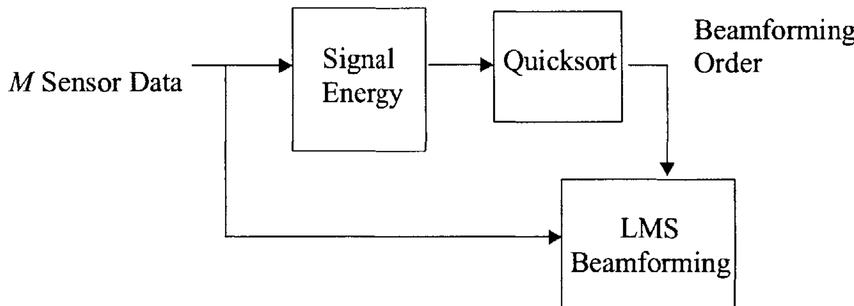


Figure 7-12: "Sort by significance" preprocessing

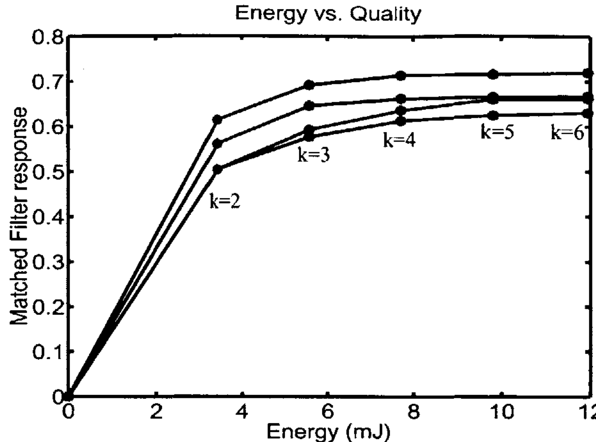


Figure 7-13: *E-Q* snapshot for Scenario 2

In Scenario 2, we apply the *most-significant-first transform* to improve the *E-Q* curves for LMS beamforming. Figure 7-13 shows the *E-Q* relationship as the source moves from location A to B. In this scenario, we can ensure that the *E-Q* graphs are always concave, thus improving *E-Q* scalability. However, there is a price to pay in computation energy. If the energy cost required to compute the correlation and quicksort was large compared to LMS beamforming, then the extra scalability is not worth the effort. However, in this case, the extra computational cost was 8.8 mJ of energy and this overhead is only 0.44% of the total energy for LMS beamforming (for the 2 sensor case).

7.4 Energy Scalability with Parallelism - Pentium Example

Parallelism results in lower power consumption due to the classic area power trade-off [10]. Doing operations in parallel implies more operations can be done in the same available time. If throughput is kept constant, each of the parallel processing blocks can be executed at a reduced frequency and voltage and the overall power is saved. In practice, having parallel datapaths increases the effective switched capacitance and interconnect overheads. In an actual adder implementation in [10], it was shown that while the switched capacitance increased by a factor of 2.15, the frequency was halved, and the voltage could be reduced by a factor of 1.7, resulting in an actual power savings of 64% from duplicating hardware.

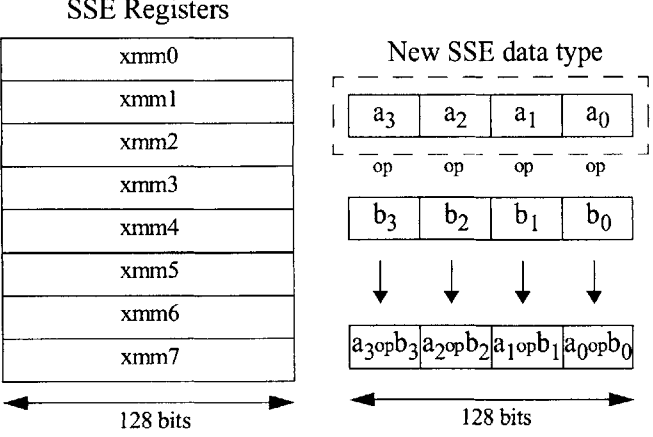


Figure 7-14: Pentium III SIMD registers and data type

The Pentium III SSE (Streaming SIMD Extensions) instructions allow for SIMD[1] operations on four single-precision floating-point numbers in one instruction as shown in Figure 7-14 [100]. We implemented some DSP library programs [101] using non-SIMD as well as the SIMD instructions. Table 7-2 shows that the average power savings possible with a combination of DVS and SIMD strategies can be almost 73%. These routines frequently appear in signal processing and multimedia applications. Based on the reduction in execution time, the clock frequency is reduced such that the latency constraints are met (i.e., throughput is still the same). The voltage reduction has been predicted using Equation 6-5. Based on the modified voltage and frequency, power savings have been estimated. It is worthwhile to note that the theoretical maximum power reduction is $(1 - 1/4^3)$ = 98.4%. However, unaligned accesses, and data rearrangements along with certain inherently non-parallelizable operations result in the obtainable power savings being 73% on an average for these programs. In practice, not all frequency levels are available and there-

1. To address the relentless demand for more computational power, microprocessor vendors have added Single Instruction Multiple Data (SIMD) capabilities and instructions to their microprocessor architectures. Most of these extensions use packed data types (such as bytes, word, quadword) and do not add new registers to the processor state. Examples include the Matrix Math Extensions (MMX) for the Intel Pentium processors [92], Multimedia Acceleration eXtensions (MAX-2) for the HP PA-RISC 2.0 [93], 3DNow! extensions to AMD K6 [94], Visual Instruction Set (VIS) for the UltraSparc [95], MIPS Digital Media Extensions and PowerPC's AltiVec technology. Although the available instructions vary, the basic idea of exploiting micro-SIMD level parallelism is common to all [96].

fore frequency quantization would lead to operating points above the optimum resulting in some reduction in power savings. Voltage conversion inefficiencies and the fact that operating voltage is not scaled in the entire processor but just the core will also reduce the power savings further.

Table 7-2: Power savings from parallel computation

Program	Time (ms)		Normalized (V_{dd}, f)		Power Savings (%)
	Normal	SIMD	f	V_{dd}	
dot	0.0022	0.0009	0.41	0.60	85.3
fir	0.3700	0.1700	0.46	0.63	81.6
exp	0.0480	0.0260	0.54	0.69	74.4
lms	1.2800	1.0900	0.85	0.89	32.2
fft	5.8000	1.7000	0.29	0.52	92.0
Average Power Reduction (%)					73.1

7.5 Summary of Contributions

We introduced the notion of energy scalable computation in the context of signal processing algorithms on general purpose processors. Algorithms that render incremental refinement of a certain quality metric such that the marginal returns from every additional unit of energy is diminishing are highly desirable in embedded applications. Using three broad classes of signal processing algorithms we demonstrated that using simple transformations (with insignificant overhead) the Energy-Quality (*E-Q*) behavior of the algorithm can be significantly improved. In general, we concluded that doing the most significant computations first enables computational energy reduction without significant hit in output quality. Finally, an energy scalable approach to computation using a combination of dynamic voltage scaling and SIMD style parallel processing was demonstrated on the Pentium III processor. Average power savings of about 73% were obtained on DSP library functions.

Chapter 8
Conclusions

This thesis is an exploration of *software* techniques to enhance the energy efficiency and therefore lifetime of energy constrained (mostly battery operated) systems. Such digital systems have proliferated into every aspect of our lives in the form of cellular phones, PDAs, laptops, MP3 players, cameras, wireless sensors, etc. Form factor, weight and battery life of these systems are often as important as the functionality offered by them. While significant research has been done on circuit design techniques for low power consumption, software optimizations have been relatively unexplored. This can partly be attributed to the fact that dedicated circuits are orders of magnitude more energy efficient that general purpose programmable solutions. Therefore, if energy efficiency was crucial, system designer opted for an Application Specific Integrated Circuit (ASIC). As such the only software power management issues that have been considered are compiler optimizations. The power benefits from pure compiler optimizations fade in contrast to the ones obtained from dedicated hardware implementations. Further, most of these optimizations are essentially performance optimizations as well.

So why should we bother with software energy efficiency? The reason is two fold

- *Technology Availability*: Of late, a variety of low power general purpose processors have entered the market and they offer sophisticated software controlled power management features such as dynamic voltage and frequency control, and a variety of sleep slates. Operating systems, therefore, have a whole new optimization space which they can exploit to improve the energy efficiency of the system.
- *Flexibility Requirement*: With constantly evolving standards and time-to-market pressure, there is a definite bias in the industry towards programmable solutions. Energy aware software techniques (beyond just compiler optimizations) are becoming as crucial as energy efficient circuit design techniques.

8.1 Contributions of this Thesis

We began our exploration with experiments involving dynamic voltage and frequency control available on the StrongARM SA-1100 processor to quantify the expected energy savings that can be obtained by exploiting the fact that power consumption depends quadratically on supply voltage and linearly on operating frequency. We obtained workload traces from different machines involved in a variety of tasks at different times of the day. From these workload traces it was apparent that the processor utilization is in fact quite low (on some machines the average was less than 10%). Therefore, substantial energy benefits could be obtained by reducing operating voltage and frequency depending on instantaneous processor utilization. However, this would involve a prior knowledge of the processor utilization profile if we were to have no visible performance loss (by slowing down the processor). We developed a simple adaptive workload filtering strategy to predict future workloads and showed that this scheme can perform really well as compared to an oracle prediction. We demonstrated that energy savings by a factor of two to three can be obtained by using our workload prediction scheme in conjunction with dynamic voltage and frequency scaling, with little performance degradation. We also defined a performance hit metric to measure the instantaneous and average performance penalty from misprediction and analyzed the effect of voltage and frequency update rates. Efficiency losses attributed to discrete voltage and frequency steps were also quantified and were shown to be within 10%.

Our workload prediction strategy can be characterized as a "best-effort" algorithm. The logical next step was to analyze dynamic voltage and frequency scaling in the context of real-time systems. We proposed the Slacked Earliest Deadling First (SEDF) scheduling algorithm and showed that it is optimal in minimizing processor energy (using DVS) and maximal lateness of a given real-time task set. Bounds on the possible energy savings, from any real-time scheduling algorithm, using dynamic voltage and frequency control were also derived. Similar results were also derived for static scheduling of periodic real-time tasks using rate monotonic theory.

Most embedded systems spend a lot of time waiting for events to process. Idle power management is in fact more important that active power management in such low duty

cycle systems. Simple idle power management techniques already exist in most battery operated systems that we use currently. Examples include simple time-out idle/sleep mechanisms present in laptop screens, palmpilots, disk drives, etc. New processors and electronic components support multiple sleep states. Standardization initiatives are on in the form of the Advanced Configuration and Power Interface (ACPI) Specification. We have demonstrated a shutdown scheme (using the μAMPS sensor node as an example) using *multiple* meaningful sleep states in the system. The concept of energy gain idle time threshold is introduced. A statistical shutdown scheme is developed that controls transitions to a set of sleep states using these idle time thresholds. Simulations and actual hardware implementations have demonstrated the efficacy of our schemes.

To quantify the efficacy of the active and idle power management schemes proposed, we implemented some of them on the μAMPS sensor node. This involved porting a popular embedded operating system (eCos) to the node and adding a power management layer to it. We demonstrated that over 50% active mode power reduction is possible for the entire node using just DVS. Using a multiple shutdown scheme idle power savings were shown to be as much as 97%. These savings could easily translate to an order of magnitude in battery life improvement depending on operational duty cycle and workload. We also quantified the energy cost of various OS kernel functions. We demonstrated that most function calls consume less than a few tens of μJoules of energy and therefore do not add a substantial overhead while providing a lot of application support.

The other focus of this thesis was energy efficiency in the application layer. The first step we took was to develop a framework for software energy estimation. Instruction current profiling techniques already exist in literature but we found that in most processors the current variation across instructions was not much. The variation in current consumption of different programs was even smaller. Therefore, we proposed a macro energy estimation technique and developed JouleTrack - a web based tool for software energy estimation. The tool predicts software current consumption within 5% of actual measurements and is an order of magnitude faster than elaborate instruction level analysis. Recognizing that leakage energy is becoming a significant factor in processors today, we also developed an elegant technique to separate leakage and switching energy components. Our microprocessor leakage model can be explained from transistor subthreshold models.

In an interesting study we demonstrated that it is possible for the leakage and switching energy components of software to become comparable in high operating voltage or low duty cycle scenarios.

An energy aware application can yield additional energy benefits and offer important trade-offs between computational quality and energy. We proposed the notion of energy-quality scalability in software. We demonstrated that using simple algorithmic transforms it is possible to restructure computations in such a way so as to reduce the loss in computational quality given a reduction in energy availability. We observed that SIMD instruction capabilities could be exploited along with dynamic voltage and frequency control for cubic reductions in power consumption (theoretically) with fixed latency requirements. We demonstrated about 70% power reduction on the Pentium III.

8.2 Future Work

Energy efficiency of software systems is a rich research space. Novel features are constantly being added to processors. In our work we could not consolidate all our ideas into one overall system and demonstrate the energy efficiency attributed to each idea. For example, while the StrongARM had dynamic voltage and frequency, it did not support SIMD arithmetic. It will be interesting to have a complete system with dynamic voltage and frequency control built into the operating system and scalable applications that exploit algorithmic transformations and SIMD features of a given target platform for scalable computation. Of particular interest would be the interplay of various energy saving techniques with each other and the overall energy savings obtained.

The standby power management scheme that we have developed uses various sleep states of the device. However, even if the device is in a given sleep state, leakage energy is getting dissipated, unless the power supply is completely turned off. Processors such as the Hitachi SH-4 increase the threshold voltage in the idle mode by biasing the substrate. This results in exponential leakage reduction. Software control of substrate biasing and power supply can hugely impact the leakage energy. Setting the inputs at various gates within the circuit to particular values can also impact the overall leakage current while the power supply is on. Software techniques to minimizing leakage energy would be another important vector to explore.

For software energy estimation it would be interesting to apply our macro estimation methodology to datapath dominant systems such as DSPs and microcontrol units which would possibly have a wider variation in instruction and program current consumption. An automated approach to estimating the model parameter would be ideal. For instance, it would be interesting to see if a set of 'training' programs could be run on a given target platform, and based on current measurements, our second order model parameters could be generated automatically.

References

[1] *The Official Bluetooth SIG Website*, http://www.bluetooth.org

[2] *3GPP - Third Generation Partnership Program*, http://www.3gpp.org

[3] A. Chandrakasan, et. al., "Design Considerations for Distributed Microsensor Systems", Custom Integrated Circuits Conference, 1999, pp. 279-286

[4] *Rechargeable Battery/Systems for Communication/Electronic Applications, An Analysis of Technology Trends, Applications, And Projected Business Climate*, NATIBO, http://www.dtic.mil/natibo/docs/BatryRpt.pdf

[5] K. Kawamoto, et. al., "Electricity Used by Office Equipment and Network Equipment in the U.S.: Detailed Report and Appendices", Ernest Orlando Lawrence Ber keley National Laboratory, http://enduse.lbl.gov/Info/LBNL-45917b.pdf

[6] *Moore's Law*, http://www.intel.com/intel/museum/25anniv/hof/moore.htm

[7] J. Eager, "Advances in Rechargeable Batteries Spark Product Innovation", Proceedings of the 1992 Silicon Valley Computer Conference, Santa Clara, Aug. 1992, pp. 243-253

[8] C. Small, "Shrinking Devices Puts the Squeeze on System Packaging", EDN 39(4), Feb. 1994, pp. 41-46

[9] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, New Jersey, 1996

[10] A. Chandrakasan, S. Sheng and R. W. Broderson, "Low-Power CMOS Design", IEEE Journal of Solid State Circuits, April 1992, pp. 472-484

[11] H. Zhang and J. Rabaey, "Low-Swing Interconnect Interface Circuits", Proceedings of the International Symposium on Low Power Electronics and Design 1998, pp. 161 - 166

[12] W. Athas, et. al., "Low Power Digital Systems Based on Adiabatic Switching Principles", IEEE Transactions on VLSI Systems, vol. 2, no. 4, Dec. 1994

[13] J. Kao, S. Narendra and A. P. Chandrakasan, "MTCMOS Hierarchical Sizing Based on Mutual Exclusive Discharge Patterns", Proceedings of the 35th Design Automation Conference, San Francisco, June 1998, pp. 495-500

[14] L. Wei, et. al., "Design and Optimization of Low Voltage High Performance Dual Threshold CMOS Circuits", Proceedings of the 35th Design Automation Conference, San Francisco, June 1998, pp. 489-494

[15] A. Chandrakasan and R. Brodersen, *Low Power CMOS Design*, IEEE Press, 1998

[16] *Intel StrongARM Processors*, http://developer.intel.com/design/strong/

[17] S. Santhanam, et al., "A Low-Cost, 300-MHz, RISC CPU with Attached Media Processor", IEEE Journal of Solid-State Circuits, vol. 33, no. 11, Nov. 1998, pp. 1829-1839

[18] *Transmeta's Crusoe Processor*, http://www.transmeta.com/crusoe/

[19] *The Technology Behind Crusoe(TM) Processors*, http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf

[20] *Hitachi SuperH RISC Engine*, http://semiconductor.hitachi.com/superh/

[21] *Texas Instruments, TMS320-C5000 Platform*, http://dspvillage.ti.com/docs/dspproducthome.jhtml

[22] *The StarCore DSP*, http://www.starcore-dsp.com/

[23] T. Xanthopoulos and A. Chandrakasan, "A Low-Power DCT Core Using Adaptive Bit-width and Arithmetic Activity Exploiting Signal Correlations and Quantizations", Symposium on VLSI Circuits, June 1999

[24] J. Goodman, A. P. Dancy, A. P. Chandrakasan, "An Energy/Security Scalable Encryption Processor Using an Embedded Variable Voltage DC/DC Converter", IEEE Journal of Solid-state Circuits, vol. 33, no. 11, Nov. 1998, pp. 1799-1809

[25] D. Stepner, N. Rajan and D. Hui, "Embedded Application Design Using a Real-Time OS", Proceedings of DAC 1999, New Orleans, pp. 151-156

[26] *MPEG Pointers and Resources*, http://www.mpeg.org/

[27] *The Wireless Application Protocol*, http://www.wapforum.org/

[28] *The MIT μAMPS Project*, http://www-mtl.mit.edu/research/icsystems/uamps/

[29] *The MEMS Clearinghouse Homepage*, http://mems.isi.edu/

[30] A. Sinha and A. Chandrakasan, "Dynamic Voltage Scheduling Using Adaptive Filtering of Workload Traces", Proceedings of the 14th International Conference on VLSI Design, Bangalore, India, Jan. 2001

[31] V. Gutnik and A. P. Chandrakasan, "An Embedded Power Supply for Low-Power DSP", IEEE Transactions on VLSI Systems, vol. 5, no. 4, Dec. 1997, pp. 425-435

[32] T. Burd, et. al., "A Dynamic Voltage Scaled Microprocessor System", International Solid State Circuits Conference 2000, pp. 294-295

[33] K. Govil, E. Chan and H. Wasserman, "Comparing Algorithms for Dynamic Speed Setting of a Low-Power CPU", Proceedings of the ACM International Conference on Mobile Computing and Networking, 1995, pp. 13-25

[34] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley and Sons, 1991

[35] MAX1717 Product Data Sheet, *Dynamically Adjustable, Synchronous Step-Down Controller for Notebook CPUs*, http://pdfserv.maxim-ic.com/arpdf/MAX1717.pdf

[36] G. Wei and M. Horowitz, "A Low Power Switching Power Supply for Self-Clocked Systems", International Symposium on Low Power Electronics and Design, 1996, pp. 313-318

[37] R. Nelson, *Probability, Stochastic Processes and Queueing Theory*, Springer-Verlag, 1995

[38] Q. Qiu and M. Pedram, "Dynamic Power Management Based on Continuous-Time Markov Decision Processes", Proceedings of the Design Automation Conference (DAC 99), New Orleans, pp. 555-561

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

[40] I. Hong and M. Potkonjak, "Power Optimization in Disk-Based Real-Time Application Specific Systems", Proceedings of the International Conference on Computer-Aided Design (ICCAD), 1996, pp. 634-637

[41] P. S. R. Diniz, *Adaptive Filtering Algorithms and Practical Implementation*, Kluwer Academic, 1997

[42] R. Min, T. Furrer and A. P. Chandrakasan, "Dynamic Voltage Scaling Techniques for Distributed Microsensor Networks", Proceedings of the IEEE Computer Society-Workshop on VLSI (WVLSI 00), April 2000

[43] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogamming in a Hard Real-Time Environment", Journal of ACM, 1973, vol. 20, no. 1, pp. 46-61

[44] W. Horn, "Some Simple Scheduling Algorithms", Naval Research Logistics Quaterly, 21, 1974

[45] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems", IEEE Software, July 1984, vol. 1, no. 3

[46] F. Yao, A. Demers and S. Shenker, "A Scheduling Model for Reduced CPU Energy", IEEE Annual Foundations of Computer Science, 1995, pp. 374-382

[47] I. Hong, M. Potkonjak and M. B. Srivastava, "On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor", Proceedings of ICCAD, 1998, pp 653-656

[48] G. Buttazzo, *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, 1997

[49] A. Sinha and A. Chandrakasan, "Energy Efficient Real-Time Scheduling", Proceedings of the International Conference on Computer Aided Design (ICCAD), San Jose, Nov. 2001

[50] M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes", Information Processing, vol. 74, 1974

[51] A. Sinha and A. Chandrakasan, "Operating System and Algorithmic Techniques for Energy Scalable Wireless Sensor Networks", Proceedings of the Second International Conference on Mobile Data Management, Hong-Kong, Jan 2001

[52] *Intel StrongARM* SA-1100 Microprocessor Developer's Manual*, http://developer.intel.com/design/strong/manuals/278088.htm

[53] J. Bray and C. Sturman, *Bluetooth - Connect Without Cables*, Prentice Hall, 2001

[54] *Advanced Configuration and Power Interface Specification*, http://www.teleport.com/~acpi

[55] W. Heinzelman, A. Chandrakasan and H. Balakrishnan, "Energy Efficient Routing Protocols for Wireless Microsensor Networks", Proceedings of 33rd Hawaii International Conference on System Sciences (HICSS 00), January 2000

[56] E. R. Dougherty, *Probability and Statistics for Engineering, Computing and Physical Sciences*, Prentice Hall 1990

[57] M. B. Srivastava, A. P. Chandrakasan and R. W. Broderson, "Predictive System Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation", IEEE Transactions on VLSI Systems, vol. 4, no. 1, March 1996, pp. 42-54

[58] L. Benini, et. al, "Policy Optimization for Dynamic Power Management", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, June 1999, pp. 813-833

[59] *The eCos Operating System*, http://www.redhat.com/ecos

[60] *Microsoft Windows CE*, http://www.microsoft.com/windows/embedded/ce/

[61] *The Palm OS Platform*, http://www.palmos.com/platform/architecture.html

[62] *The μITRON API*, http://sources.redhat.com/ecos/docs-latest/ref/ecos-ref.a.html

[63] *The EL/IX Homepage*, http://sources.redhat.com/elix/

[64] *eCos Downloading and Installation*, http://sources.redhat.com/ecos/getstart.html

[65] *The GNU Project*, http://www.gnu.org/

[66] Intel Corp., *Intel StrongARM SA-1110 Microprocessor - Advanced Developer's Manual*, June 2000

[67] *Intel StrongARM SA-1110 Linecard*, http://developer.intel.com/design/strong/linecard/sa-1110/

[68] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Feb. 2001

[69] *eCos Reference Manual*, http://sources.redhat.com/ecos/docs-latest/pdf/ecos-ref.pdf

[70] *eCos User's Guide*, http://sources.redhat.com/ecos/docs-latest/pdf/user-guides.pdf

[71] *The Intel XScale Microarchitecture*, http://developer.intel.com/design/intelxscale/

[72] *μAMPS Operating System and Software Homepage*, http://gatekeeper.mit.edu

[73] *JouleTrack - A Web Based Tool for Software Energy Profiling*, http://dry-martini.mit.edu/JouleTrack/

[74] A. Sinha and A. Chandrakasan, "JouleTrack - A Web Based Tool for Software Energy Profiling", Proceedings of the 38th Design Automation Conference, Las Vegas, June 2001

[75] V. Tiwari and S. Malik, "Power Analysis of Embedded Software: A First Approach to Software Power Minimization", IEEE Transactions on VLSI Systems, vol. 2, Dec. 1994

[76] Advanced RISC Machines Ltd., *Advance RISC Machines Architectural Reference Manual*, Prentice Hall, New York, 1996

[77] Advanced RISC Machines Ltd., *ARM Software Development Toolkit Version 2.11 : User Guide*, May 1997

[78] *Solution Engine*, http://semiconductor.hitachi.com/tools/solution_engine.html

[79] J. Montanaro, et. al., "A 160MHz 32b 0.5W CMOS RISC Microprocessor", IEEE Journal of Solid State Circuits, November 1996, pp. 1703-1714

[80] J. Sheu, et. al., "BSIM: Berkeley Short-Channel IGFET Model for MOS Transistors", IEEE Journal of Solid-State Circuits, SC-22, 1987

[81] R. X. Gu and M. I. Elmasry, "Power Dissipation Analysis and Optimization of Deep Submicron CMOS Digital Circuits", IEEE Journal of Solid State Circuits, vol. 31, no. 5, May 1996, 707-713

[82] S. H. Nawab, et. al., "Approximate Signal Processing", Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, vol. 15, no. 1/2, Jan. 1997, pp. 177-200

[83] A. Sinha and A. Chandrakasan, "Energy Aware Software", Proceedings of the XIII International Conference on VLSI Design, Calcutta, India, Jan. 2000

[84] A. Sinha, A. Wang, and A. Chandrakasan, "Algorithmic Transforms for Efficient Energy Scalable Computation," The 2000 IEEE International Symposium on Low-Power Electronic Design (ISLPED 00), Italy, August 2000

[85] A. V. Oppenheim and R. W. Schafer, *Discrete Time Signal Processing*, Prentice Hall, New Jersey, 1989

[86] M. Mehendale, A. Sinha and S. D. Sherlekar, "Low Power Realization of FIR Filters Implemented Using Distributed Arithmetic", Proceedings of Asia South Pacific Design Automation Conference, Yokohama, Japan, Feb. 1998

[87] N. Ahmed, T. Natarajan and K. R. Rao, "Discrete Cosine Transform", IEEE Transactions on Computers, vol. 23, Jan. 1974, pp. 90-93

[88] W. H. Chen, C. H. Smith and S. C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform", IEEE Trans. on Communication, vol. 25, Sept 1977, pp. 1004-1009

[89] L. McMillan and L. A. Westover, "A Forward-Mapping Realization of the Inverse Discrete Cosine Transform", Proceedings of the Data Compression Conference (DCC 92), March 1992, pp. 219-228

[90] Yao, et. al., "Blind Beamforming on a Randomly Distributed Sensor Array System", IEEE Journal on Selected Areas in Communications, vol. 16, no. 8, Oct. 1998, pp. 1555-1567

[91] A. Wang, W. Heinzelman and A. Chandrakasan, "Energy-Scalable Protocols for Battery-Operated Microsensor Networks", Proceedings of the IEEE workshop on IEEE Workshop on Signal Pprocessing Systems (SIPS 99), Oct. 1999

[92] A. Peleg and U. Weiser, "MMX Technology Extensions to the Intel Architecture", IEEE Micro, vol. 16, no. 4, Aug. 1996, pp. 42-50

[93] R. B. Lee, "Subword Parallelism with MAX-2", IEEE Micro, vol. 16, no. 4, Aug. 1996, pp. 51-59

[94] *AMD-3DNOW! Technology Manual, Instruction Set Architecture Specification*, http://www.amd.com/K6/k6docs/

[95] M. Tremblay et.al., "VIS Speeds New Media Processing", IEEE Micro, vol. 16, no. 4, Aug. 1996, pp. 10-20

[96] M. D. Jennings and T. M. Conte, "Subword Extensions for Video Processing on Mobile Systems", IEEE Concurrency, July-Sept. 1998, pp. 13-16

[97] *Intel SpeedStep Technology*, http://www.intel.com/mobile/pentiumIII/ist.htm

[98] *Transmeta Crusoe LongRun Technology*, http://www.transmeta.com/crusoe/low-power/longrun.html

[99] *AMD K6 PowerNOW*, http://www.amd.com/products/cpg/mobile/powernow.html

[100] *Intel Pentium III SIMD Extensions*, http://developer.intel.com/vtune/cbts/simd.htm

[101] *TMS320C54x DSP Function Library*, http://www.ti.com/sc/docs/products/dsp/c5000/c54x/54dsplib.htm

Appendix A

Operating System Energy Characterization

Table A.1 describes the timing overhead and average energy cost for various operating system kernel calls on the μAMPS sensor node. Key operations in the kernel were measured by using a simple test program which exercises the various kernel primitive operations. A hardware timer which drives the real-time clock, was used for these measurements. This timer can be read with quite high resolution. For each measurement, the operation was repeated a number of times. Time stamps were obtained directly before and after the operation was performed. The data gathered for the entire set of operations was then analyzed, generating average, maximum and minimum values. The sample variance (a measure of how close most samples are to the mean) was also calculated. The cost of obtaining the real-time clock timer values was also measured, and was subtracted from all measured times. The average energy cost was computed by multiplying the average current consumption with the average estimated execution time. Most kernel functions can be measured separately. In each case, a reasonable number of iterations are performed. Where the test case involves a kernel object, for example creating a task, each iteration is performed on a different object. There is also a set of tests which measures the interactions between multiple tasks and certain kernel primitives. Most functions are tested in such a way as to determine the variations introduced by varying numbers of objects in the system. For example, the mailbox tests measure the cost of a `peek` operation when the mailbox is empty, has a single item, and has multiple items present. In this way, any effects of the state of the object or how many items it contains can be determined. There are a few things to consider about these measurements. Firstly, they are quite micro in scale and only measure the operation in question. These measurements do not adequately describe

how the timings would be perturbed in a real system with multiple interrupting sources. Secondly, the possible aberration incurred by the real-time clock is explicitly avoided. Virtually all kernel functions have been designed to be interruptible. Thus the times presented are typical, but best case, since any particular function may be interrupted by the clock tick processing. This number is explicitly calculated so that the value may be included in any deadline calculations required by the end user. Lastly, the reported measurements were obtained from a kernel and sensor node with all options at their default values.

Table A.1: Energy characterization of various OS function calls

Kernel Function	**Time (μs)**				**Avg Energy (μJ)**
	Avg	**Min**	**Max**	**Var**	
THREADS					
Create thread	10.02	8.14	23.60	1.13	5.61
Yield thread [all suspended]	1.46	1.09	11.39	0.40	0.82
Suspend [suspended] thread	2.08	1.63	7.87	0.39	1.16
Resume thread	1.13	0.81	5.15	0.16	0.63
Set priority	1.57	1.36	2.98	0.13	0.88
Get priority	0.42	0.00	1.36	0.31	0.24
Kill [suspended] thread	4.81	3.80	23.60	0.82	2.69
Yield [no other] thread	1.35	1.09	7.60	0.26	0.76
Resume [suspended low prio] thread	3.01	2.44	7.87	0.61	1.69
Resume [runnable low prio] thread	1.07	0.81	2.44	0.07	0.60
Suspend [runnable] thread	2.16	1.63	3.80	0.39	1.21
Yield [only low prio] thread	1.30	1.09	3.53	0.16	0.73
Suspend [runnable->not runnable]	1.09	0.81	1.90	0.03	0.61
Kill [runnable] thread	3.99	3.80	7.32	0.28	2.23
Destroy [dead] thread	5.30	3.26	20.35	0.96	2.97
Destroy [runnable] thread	6.43	5.70	11.94	0.58	3.60
Resume [high priority] thread	15.24	12.21	32.28	1.82	8.53
Thread switch	3.20	2.98	12.75	0.21	1.79

Table A.1: Energy characterization of various OS function calls

Kernel Function	Time (μs)				Avg Energy (μJ)
	Avg	Min	Max	Var	
SCHEDULER					
Scheduler lock	0.23	0.00	3.26	0.11	0.13
Scheduler unlock [0 threads]	0.75	0.54	1.36	0.10	0.42
Scheduler unlock [1 suspended]	0.75	0.54	0.81	0.10	0.42
Scheduler unlock [many suspended]	0.75	0.54	1.63	0.10	0.42
Scheduler unlock [many low prio]	0.75	0.54	1.36	0.10	0.42
MUTEX					
Init mutex	0.49	0.27	4.34	0.28	0.27
Lock [unlocked] mutex	1.99	1.09	11.66	0.64	1.11
Unlock [locked] mutex	1.95	1.36	14.38	0.81	1.09
Trylock [unlocked] mutex	1.24	1.09	4.88	0.26	0.69
Trylock [locked] mutex	0.81	0.81	0.81	0.00	0.45
Destroy mutex	0.49	0.27	2.98	0.21	0.27
Unlock/Lock mutex	7.45	7.32	11.12	0.24	4.17
MAILBOX					
Create mbox	0.77	0.54	5.15	0.30	0.43
Peek [empty] mbox	1.14	0.81	1.90	0.19	0.64
Put [first] mbox	1.94	1.36	12.21	0.83	1.09
Peek [1 msg] mbox	0.19	0.00	0.27	0.11	0.11
Put [second] mbox	1.40	1.36	2.17	0.08	0.78
Peek [2 msgs] mbox	0.21	0.00	1.09	0.13	0.12
Get [first] mbox	1.86	1.36	11.94	0.76	1.04
Get [second] mbox	1.40	1.36	2.17	0.08	0.78
Tryput [first] mbox	1.54	1.36	7.32	0.36	0.86
Peek item [non-empty] mbox	1.54	1.09	6.51	0.49	0.86
Tryget [non-empty] mbox	1.56	1.36	7.87	0.39	0.87

Table A.1: Energy characterization of various OS function calls

Kernel Function	Time (μs)				Avg Energy (μJ)
	Avg	Min	Max	Var	
Peek item [empty] mbox	1.16	1.09	1.90	0.12	0.65
Tryget [empty] mbox	1.23	1.09	1.90	0.15	0.69
Waiting to get mbox	0.32	0.27	1.90	0.10	0.18
Waiting to put mbox	0.30	0.27	1.09	0.05	0.17
Delete mbox	1.75	1.36	9.77	0.50	0.98
Put/Get mbox	4.33	4.07	10.85	0.49	2.42
SEMAPHORES					
Init semaphore	0.47	0.27	2.71	0.20	0.26
Post [0] semaphore	1.31	0.81	4.61	0.42	0.73
Wait [1] semaphore	1.24	1.09	3.53	0.21	0.69
Trywait [0] semaphore	1.08	0.81	4.07	0.20	0.60
Trywait [1] semaphore	1.08	0.81	1.09	0.02	0.60
Peek semaphore	0.40	0.27	1.90	0.17	0.22
Destroy semaphore	0.51	0.27	2.71	0.18	0.29
Post/Wait semaphore	4.63	4.34	13.56	0.56	2.59
COUNTERS					
Create counter	0.69	0.27	6.24	0.35	0.39
Get counter value	0.84	0.00	3.53	0.62	0.47
Set counter value	0.26	0.00	1.63	0.10	0.15
Tick counter	1.38	1.36	2.17	0.05	0.77
Delete counter	0.42	0.27	1.90	0.18	0.24
ALARMS					
Create alarm	0.91	0.54	4.34	0.21	0.51
Initialize alarm	3.60	2.44	18.45	1.37	2.02
Disable alarm	0.29	0.00	2.44	0.13	0.16
Enable alarm	2.00	1.90	5.15	0.20	1.12

Table A.1: Energy characterization of various OS function calls

Kernel Function	Time (μs)				Avg Energy (μJ)
	Avg	Min	Max	Var	
Delete alarm	0.61	0.54	2.71	0.13	0.34
Tick counter [1 alarm]	1.94	1.63	6.24	0.27	1.09
Tick counter [many alarms]	15.99	15.73	17.63	0.13	8.95
Tick & fire counter [1 alarm]	2.64	2.44	4.34	0.18	1.48
Tick & fire counters [>1 together]	42.34	42.32	43.13	0.05	23.71
Tick & fire counters [>1 separately]	16.83	16.55	17.63	0.05	9.42
Alarm latency [0 threads]	3.90	3.80	12.48	0.19	2.18
Alarm latency [2 threads]	4.51	3.80	11.12	0.53	2.53
Alarm latency [many threads]	26.88	23.06	34.99	2.57	15.05
Alarm -> thread resume latency	9.18	8.68	64.83	0.96	5.14

Appendix B

Current Measurements

B.1 StrongARM SA-1100 Instruction Current Consumption

The following table lists the instruction current consumption for the SA-1100 measured on the Brutus Evaluation platform at an operating voltage of 206MHz and core supply voltage of 1.5 V. Each instruction current profile is averaged over its various addressing modes and variations.

Table B.1: SA-1100 Instruction Current Consumption

Instruction	Average Curent (A)
ADC	0.174
ADD	0.178
AND	0.174
BIC	0.174
CMN	0.180
CMP	0.180
EOR	0.179
MOV	0.176
MVN	0.176
ORR	0.179
RSB	0.181
RSC	0.181
SBC	0.180
SUB	0.179
TEQ	0.179
TST	0.206

Table B.1: SA-1100 Instruction Current Consumption

Instruction	Average Curent (A)
MLA	0.206
MUL	0.185
LDR	0.208
LDRB	0.208
LDRBT	0.185
LDRT	0.185
STR	0.230
STRB	0.228
STRBT	0.229
STRT	0.230
LDM	0.237
STM	0.228
MRS	0.174
MSR	0.169
SWP	0.165
B	0.169
NOP	0.172

B.2 Hitachi SH7750 Instruction Current Consumption

The following table shows the average core current consumption for the Hitachi SH7750 processor (SH-4) measured using the SolutionEngine development board. The core supply voltage was 1.95 V and the processor was running at 200 MHz. This list is not exhaustive and excludes the floating point instructions.

Table B.2: SH7750 Instruction Current Consumption

Instruction	Average Current (A)
MOV	0.2388
SWAP	0.2676
ADD	0.2827

Table B.2: SH7750 Instruction Current Consumption

Instruction	Average Current (A)
CMP	0.3034
DIV	0.3057
MAC	0.3486
SUB	0.2861
AND	0.2762
OR	0.2766
XOR	0.2819
TST	0.3014
ROT	0.2819
SHA	0.2798
SHL	0.2751
B	0.3339
BRA	0.3261
JMP	0.2610
CLRMAC	0.2805
SETT/CLRT	0.3019
NOP	0.2968

B.3 Program Current Consumption

The following table lists the program current consumption measured for a set of DSP routines at optimal voltage frequency points on the SA-1100 processor.

Table B.3: Program Currents on SA-1100

Frequency (MHz)	Programs					
	dct	dhry	sort	log	fir	fft
206	0.2335	0.2474	0.2309	0.2345	0.2443	0.2338
192	0.2013	0.2134	0.1984	0.2025	0.2106	0.2019
177	0.1713	0.1817	0.1681	0.1725	0.1795	0.1719

Table B.3: Program Currents on SA-1100

Frequency (MHz)	Programs					
	dct	dhry	sort	log	fir	fft
162	0.1441	0.1529	0.1430	0.1451	0.1510	0.1441
148	0.1256	0.1335	0.1236	0.1266	0.1318	0.1261
133	0.1085	0.1153	0.1073	0.1094	0.1138	0.1089
118	0.0923	0.0981	0.0910	0.0931	0.0969	0.0927
103	0.0731	0.0777	0.0715	0.0736	0.0767	0.0734
89	0.0598	0.0636	0.0596	0.0603	0.0627	0.0600
74	0.0475	0.0505	0.0473	0.0479	0.0498	0.0477
59	0.0378	0.0402	0.0373	0.0381	0.0396	0.0379

Appendix C

μAMPS Webtool

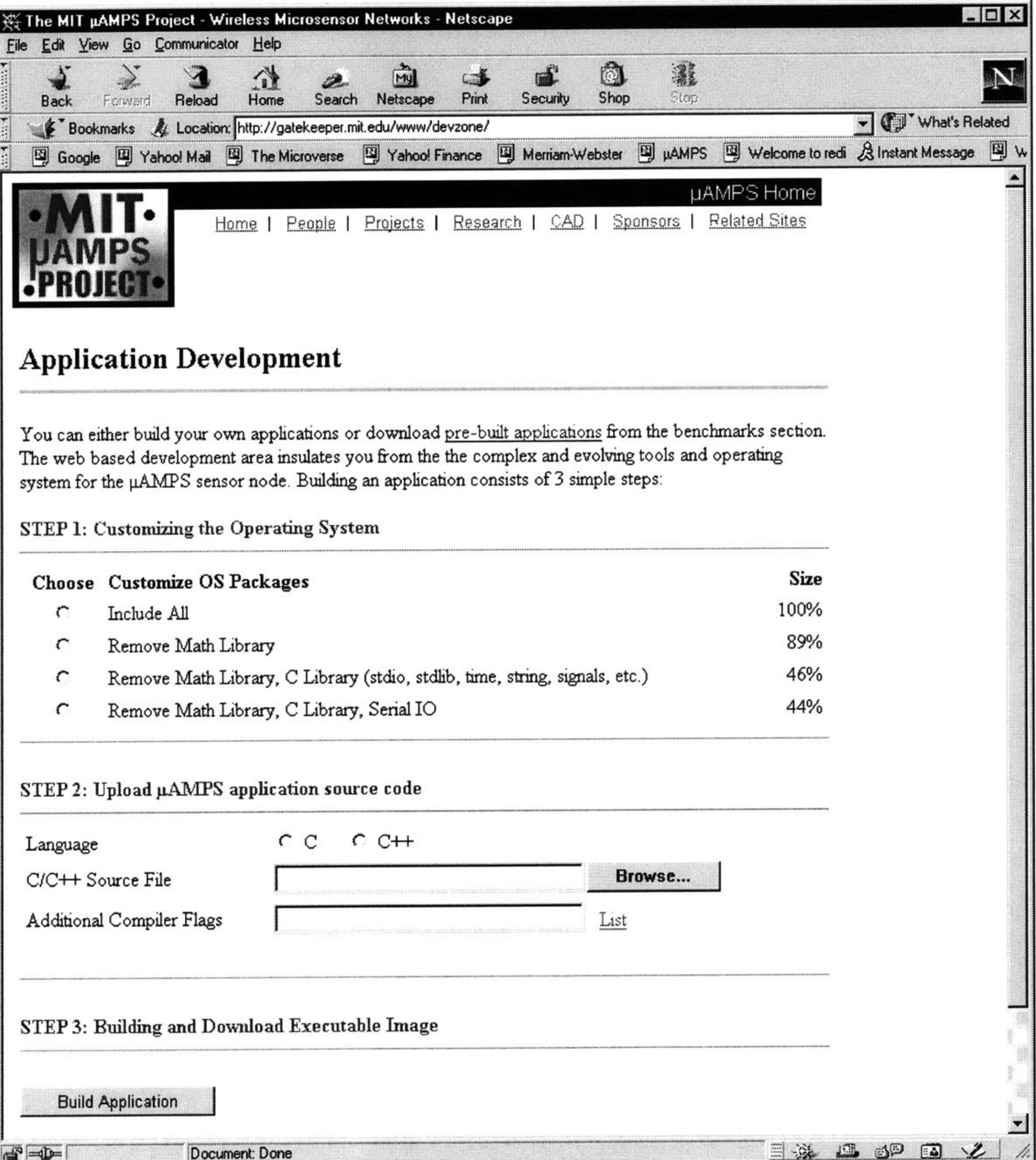


Figure C-1: μAMPS operating system and software webtool

Appendix D

JouleTrack Screenshots

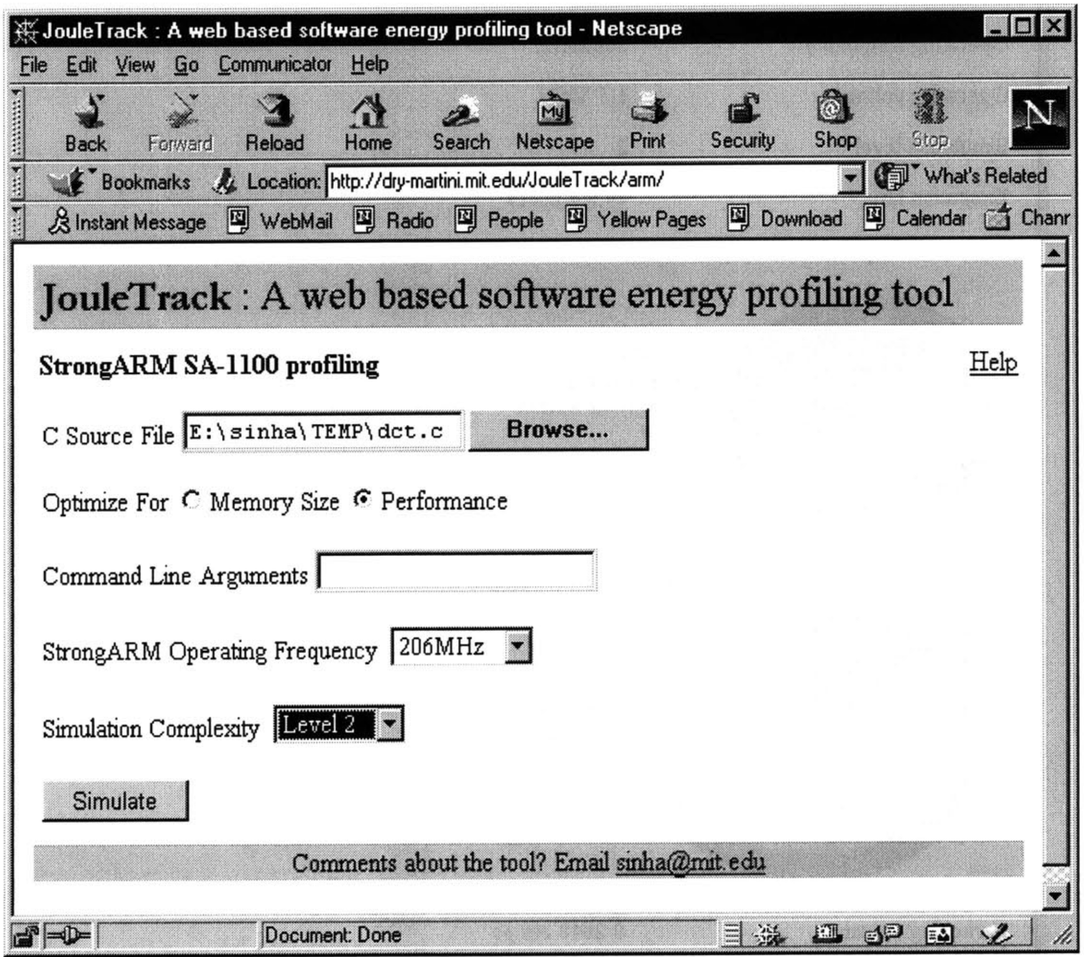


Figure D-1: JouleTrack remote file upload and operating point selection

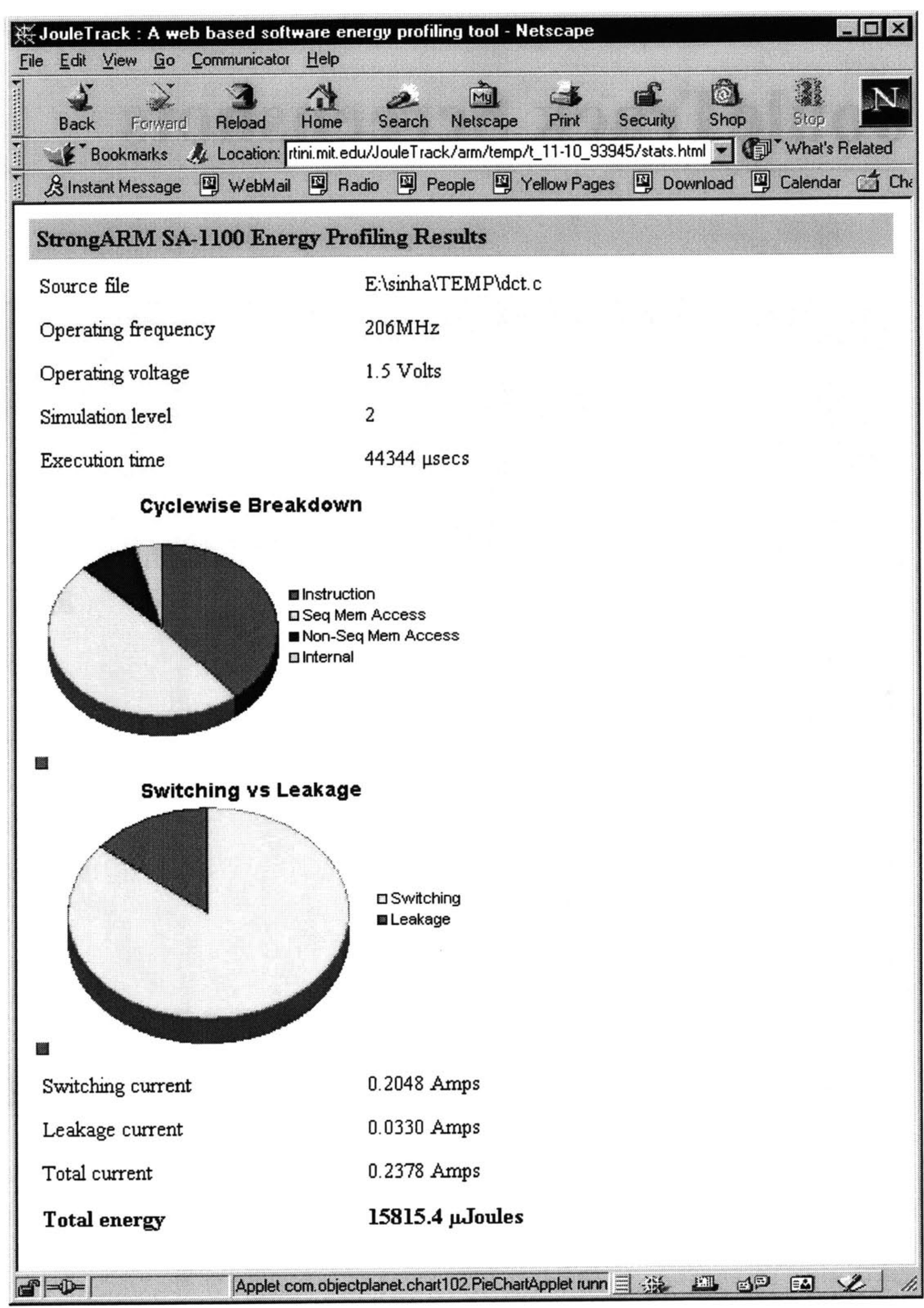


Figure D-2: JouleTrack energy profiling output

Appendix E

Scalable Image Decoding

Table E.1 compares the energy scalability of image decoding using Chen's algorithm and the FM-IDCT algorithm. It can be easily seen that the FM-IDCT is highly energy scalable.

Table E.1: Scalability in Image Decoding

Energy Fraction	Chen's Algorithm	FM-IDCT
12.5%		
25%		
37.5%		

Table E.1: Scalability in Image Decoding

Energy Fraction	Chen's Algorithm	FM-IDCT
50%		
62.5%		
75%		
87.5%		

Table E.1: Scalability in Image Decoding

Energy Fraction	Chen's Algorithm	FM-IDCT
100%		