NORTHWESTERN UNIVERSITY

Holistic Computer Architectures based on Application, User, and Process Characteristics

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By
Arindam Mallik

EVANSTON, ILLINOIS
June 2008

© Copyright by Arindam Mallik 2008
All Rights Reserved

Abstract

Holistic Computer Architectures based on Application, User, and Process Characteristics

Arindam Mallik

As we move into deeper sub-micron technologies, the complexity of pushing the circuit performance further is becoming an important obstacle. To achieve better performance, there is an increasing need for collaboration of higher level (e.g. microarchitecture-level) and circuit level optimizations. Traditionally for a computer system, applications lie at the top of the whole spectrum. Previously, researchers have looked into application characteristics to optimize the performance of the system. The ever-increasing need for improvement in system performance and power utilization led me to believe that we need to look beyond the application level to utilize the system resources more intelligently. Note that, the primary objective of a computing system is to satisfy the user's expectations. Previous researchers have worked into user satisfaction while interacting with a system. But their objective was to dynamically optimize the operating system behavior to satisfy the user. We believe that including individual user's preferences to optimize system hardware utilization would lead to better performance and power. The results presented in this work support this hypothesis. Additionally, analysis of the materials that lies at the lowermost end of the system spectrum

opens up a number of opportunities to optimize the system performance. Every individual piece of hardware possesses unique properties even after going through the same manufacturing technology. Therefore, the "one-size-fits-all" approach of current DVFS schemes is suboptimal in the presence of process variations. We have proposed architectural optimization based on process characteristics of individual CPU. Circuit designers typically consider the worst case scenario to predict the default voltage properties of a processor chip. The hard constraint of reliability has created a gap between the default value and the threshold where a circuit can work flawlessly. We have shown that treating the correctness as an objective can improve the system performance with noted reductions in power consumption. The results obtained from these research works have led us to propose the idea of the "holistic architecture".

Acknowledgements

"Learn from yesterday, live for today, hope for tomorrow. The important thing is to not stop questioning. Curiosity has its own reason for existing."

- Albert Einstein

Research to me is a lifelong journey en route to the source of light of knowledge. A journey is easier when we travel together. Interdependence is certainly more valuable than independence. This dissertation is the result of four years of work whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

First and foremost, I would like to thank my advisor, Gokhan Memik. Gokhan, your invaluable suggestions, pointed directions and lucid explanations have helped me in every single step while working on this dissertation. You are my constant source of inspiration. You have shown me the light whenever I seem to have stuck in a dark corner. You always seem to have simple answers to problems that looked so complicated to me. I am really fortunate to have you by my side for the entire journey.

I would also like to thank the other members of my PhD committee who have collaborated with me and took effort in reading and providing me with

valuable comments on this thesis: Robert P. Dick, Peter A. Dinda, Yehea Ismail, Russ Joseph and Seda Memik. I thank you all.

Thanks are due for Bin Lin and Jack Cosgrove with whom I have collaborated while working on this dissertation. I would like to use this opportunity to thank my fellow labmates - Abhishek, Berkin, Matt, Prabhat, Serkan, and Yu. For this research, data were essential. I collected *a lot* of data. They helped me wholeheartedly. My roommates and friends at Northwestern deserve thanks for helping me to cruise through the graduate life –Debjit, Peter, Paramita, Rubina, Shantanu, and Som.

My journey to this stage of the academic life has been motivated with the guidance and support of some wonderful persons I have come across my life. I express deep gratitude to Ajit Ghatak and Ajay Bhattacharya – two finest mentors. I am also thankful to all my other teachers, whose names I could not include here.

I should not have been writing this dissertation without my baba, Aniruddha Mallik who formed part of my vision and taught me the good things that really matter in life. I remember his exact words after my high school graduation, “If you aim to climb Mount Everest, you would at least reach The Kanchanjungha”. I fondly remember those starry nights at our rooftop when we shared many of our dreams. I wish he would have been here to share this golden moment with me. The seeds that he had sown in my mind have been thoroughly

nurtured by my mom, Maya and didi, Ankita. I am really proud to have them as my family. I am very grateful to my fiancée Rima, for her unconditional love and relentless support during the PhD period. She did not complain about my 'last few weeks of dissertation' which actually lasted almost a year. My life would have been incomplete without her in my side all these years.

Now it is time to thank the abstract objects and other forms of life who have contributed greatly for all these years. Thanks to my cubicle-mate, a goldfish called Jharna has been constantly vigilant on my work. I would like to thank the city of Evanston, for the wonderful life it provided outside my lab. I would like to thank my favorite Parker pen that has been my constant companion since the high school days.

Finally, I *pranaam* God – Father, Son and Spirit - for His unconditional love, guidance, sustenance, and strength. He is the source of all my inspiration, dedication and strength to complete this work.

Arindam Mallik

Northwestern University
October, 2007

To the loving memory of my father

Contents

CHAPTER 1 15
Introduction to Holistic Architecture 15
1.1. Motivation for a holistic architecture 16
1.2. Correctness-aware Application Adaptive Execution 20
1.3. Task Allocation based on statistical variation 21
1.4. User-Experience Driven Optimizations 21
1.5. User-Perceived Performance Evaluation 22
1.6. Process-aware Voltage Setting 23
1.7. Dissertation Overview 24
CHAPTER 2 25
Application-Level Error Measurements for Application Specific Processors 25
2.1. Analytical Model of Error Probability 28
2.2. Sources of Errors 36
2.3. Error Classification 37
2.4. Applications and Error Metrics 38
2.5. Error Injection and Measurement 42
2.6. Simulation Environment 45
2.7. Simulation Results 46
2.8. Related Work 52
CHAPTER 3 54
Reliability-Performance Tradeoff Analysis 54
A. Clumsy Processing in Packet Processors 56
3.1. Introduction 56
3.2. Applications and Error Measurement 60
3.3. Clock Variation and Fault Detection 61
3.4. Experimental Results 69
3.5. Previous Work on Resilient Architectures 81
B. Statistical Task Allocation in Multicore Network Processors 83
3.6. Modularity in Network Applications 87
3.7. Implementation Gap 89

3.8. Implementation Gap Closure Approaches 90
3.9. Applications 94
3.10. Probability Distribution of Packets 96
3.11. Statistical Task Allocation in NPs 100
3.12. Experiments 107
3.13. Related Work on Task Allocation 113
3.14. Conclusions 115
CHAPTER 4 117
User-Directed Power Management 117
4.1. User-Driven Frequency Scaling 121
4.2. Process-Driven Voltage Scaling 130
4.3. Evaluation 135
4.4. Discussion 151
4.5. Related Work 155
4.6. Conclusion 158
CHAPTER 5 159
User-perceived performance evaluation 159
5.1. User-Perceived Performance 162
5.2. PICSEL Framework 168
5.3. Evaluation 175
5.4. Related Work 192
5.5. Conclusion 195
CHAPTER 6 196
Contributions and Conclusions 196
REFERENCES 200

List of Figures

Figure 1.1. Abstraction levels in traditional computer architecture 17

Figure 1.2. Abstraction levels in holistic computer architecture. 19

Figure 2.1. Voltage at a circuit node at two different frequencies 29

Figure 2.2. Decrease of voltage swing with increase of frequency 29

Figure 2.3. A simple D Flip-Flop 30

Figure 2.4. Noise immunity curves of a D flip-flop at various voltage swings 30

Figure 2.5. Noise amplitude at various switching combination of neighboring lines of a victim line 32

Figure 2.6. Probability of error at different cycle time 34

Figure 2.7. Probability of error at various voltage swings 34

Figure 2.8. Error Generation probability for Route application 48

Figure 2.9. Error Generation probability for DRR application 48

Figure 2.10. Error Generation probability for TL application 48

Figure 2.11. Error Generation probability for NAT application 49

Figure 2.12. Error Generation probability for MD5 application 49

Figure 2.13. Error Generation probability for CRC application 49

Figure 2.14. Error Generation probability for URL application 50

Figure 2.15. Fatal Error probability for different applications 52

Figure 3.1. Transistor Integration Capacity 55

Figure 3.2. Error Probability of ROUTE application 71

Figure 3.3. Error Probability of NAT application 72

Figure 3.4. Fatal error probabilities for different clock rates. 74

Figure 3.5. Energy-delay2-fallibility2 product for the simulated configurations, the dynamic configuration, and the static configuration with Cr = 1, 0.75, 0.5, and 0.25 for the ROUTE application. The bars represent the relative energy-delay2-fallibility2 product with respect to Cr = 1 with no-detection. 77

Figure 3.6. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the CRC application. 77

Figure 3.7. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the MD5 application. 77

Figure 3.8. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the TL application. 78

Figure 3.9. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the DRR application. 78

Figure 3.10. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the NAT application. 78

Figure 3.11. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the URL application. 79

Figure 3.12. Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 the average application. 79

Figure 3.13. Click configuration for TTL decrement 89

Figure 3.14. Implementation Gap 91

Figure 3.15. Click configuration tree for the IPV4Router application 95

Figure 3.16. Illustration of module distribution in IPV4Router application 102

Figure 3.17. Processor throughput for DRR application 110

Figure 3.18. Processor throughput for RED application 110

Figure 3.19. Processor throughput for Home_Node application 110

Figure 3.20. Processor Throughput in Route application 110

Figure 3.21. Resource Utilization in DRR application 112

Figure 4.1. User pessimism. 124

Figure 4.2. The frequency for UDFS schemes during FIFA game for a representative user. 128

Figure 4.3. Frequency over time for UDFS1 aggregated over 20 users. 138

Figure 4.4. Frequency over time for UDFS2 aggregated over 20 users. 139

Figure 4.5. Comparison of UDFS algorithms, UDFS+PDVS, and Windows XP DVFS (CPU Dynamic Power). Chebyshev bound-based (1-p) values for difference of means from zero are also shown. 143

Figure 4.6. System Power Measurement Setup 146

Figure 4.7. Comparison of UDFS algorithms, UDFS+PDVS, and Windows XP DVFS (measured system power with display off). Chebyshev bound-based (1-p) values for difference of means from zero are also shown. 148

Figure 4.8. Mean and peak temperature measurement. 150

Figure 4.9. Power improvement in the multitasking environment. Chebyshev bound-based (1-p) values for difference of means from zero are also shown. 155

Figure 5.1. IPS and APC curve 165

Figure 5.2. APR curves for the three applications 167

Figure 5.3. Graphics pipeline in a modern PC 169

Figure 5.4. Frequency state diagram 180

Figure 5.5. The CPU dynamic power reduction with cPICSEL and aPICSEL over Windows DVFS 181

Figure 5.6. System power measurement setup 184

Figure 5.7. The system power reduction with cPICSEL and aPICSEL over Windows DVFS 185

Figure 5.8. Peak temperature reduction. 188

Figure 5.9. User ranking distribution. 189

Figure 5.10. Thermal emergency under Windows DVFS 192

List of Tables

Table 2-A. NetBench Applications and Their Properties 40

Table 2-B. Fallibility Factor of Different Applications 51

Table 3-A. Fallibility Factor of Different Applications 74

Table 3-B. Important characteristics of representative Network Processor Designs: exec. cores is the number of execution cores, and parallelism technique is the technique(s) used for task or instruction level parallelism (MT: Multi-Threading, VLIW: Very-Long Instruction Word) in the execution cores 88

Table 3-C. Probability Distribution of IPV4Router Elements 97

Table 3-D. Probability Distribution of Application Elements 98

Table 3-E. Probability Distribution of IPV4Router Stages 100

Table 4-A. Minimum stable V_{dd} for different operating frequencies and temperatures 133

Table 4-B. Power reduction for Windows DVFS and DVFS+PDVS 141

Table 4-C. Average number of user events. 151

Table 4-D. Number of voltage transitions 153

Table 5-A. User-Perceived Performance Metrics 165

CHAPTER 1

Introduction to Holistic Architecture

Computer architecture is the science and art of selecting and interconnecting hardware components to create computers that meet functional performance and cost goals. In computer engineering, it is the conceptual design and fundamental operational structure of a computer system. Computer architecture is a blueprint and functional description of requirements (especially speeds and interconnections) and design implementations for the various parts of a computer focusing largely on the way by which the central processing unit (CPU) performs internally and accesses addresses in memory.

The exact form of a computer system depends on the constraints and goals for which it was optimized. Computer architectures usually trade off standards, cost, memory capacity, latency and throughput. Sometimes other considerations, such as features, size, weight, reliability, expandability and power consumption are factors as well.

The most common optimization scheme carefully chooses the bottleneck that most reduces the computer's speed. Ideally, the cost is allocated

proportionally to assure that the data rate is nearly the same for all parts of the computer, with the most costly part being the slowest. This is how skillful commercial integrators optimize personal computers.

1.1. Motivation for a holistic architecture

Effectively, computer architecture serves as an interface between technology trends and marketplace demands. It delivers a computing system optimized as per the needs of the industry a.k.a. the users. Over the years, the optimizations objectives have been changed based on the innovations in the field of IC technology. During the 80's, area optimization was the main research objective for both academic and industrial researchers [1, 2]. As a result, innovations in computer architecture resulted in chips optimized for area. Over the 90's, power has been the key bottleneck for state of the art technologies [3, 4]. Subsequently, architectures proposed over that decade have been primarily focused towards low power solutions. As we moved into the new century, reliability has been detected as one of the primary bottlenecks for improving system performance [5-8]. As a result, we observe a trend towards reliability-aware architectures in the last few years. Hence, the design challenges in computer architecture has mutated over time. The computer architects constantly explore new ways to satisfy the market demand.

The innovations in computer architecture and progress of Silicon manufacturing technology are closely inter-related [9]. Constant improvements in CMOS technology since the 70's has helped the computer architects to come up

with faster, denser, cooler and cheaper computing systems. However, we have reached an important juncture of technological innovation history where traditional architectural innovations are facing an inevitable halt due to inherent changes in the manufacturing technology. In this dissertation, we have proposed additional abstraction layers that can result in system architecture that is not possible otherwise.

Traditionally, a computer system is usually represented as consisting of five abstraction levels: hardware, firmware, assembler, operating system and applications [10]. Figure 1.1 presents the organization of abstraction layers in traditional computer architecture.

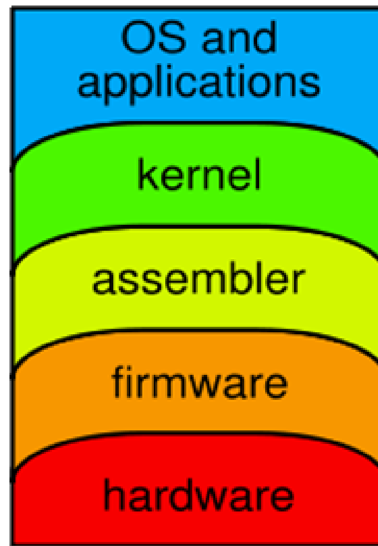


Figure 1.1. *Abstraction levels in traditional computer architecture*

As we move into deeper sub-micron technologies, the complexity of pushing the circuit performance further is becoming an important obstacle [11, 12]. To achieve better performance, there is an increasing need for collaboration of higher

level (e.g. microarchitecture-level) and circuit level optimizations [13, 14]. Traditionally for a computer system, applications lie at the top of the whole spectrum [10]. Previously, researchers have looked into application characteristics to optimize the performance of the system [15, 16]. The ever-increasing need for improvement in system performance and power utilization led me to believe that we need to look beyond the application level to utilize the system resources more intelligently. My research questions this fundamental definition about computer system. We propose a holistic computer architecture that considers two new layers lying at two extreme ends of the current set of abstraction levels – users and materials. Users lie at the top of the abstraction levels interacting directly with the applications. On the other hand, due to process variation every individual processor shows a variation from the default behavior specified by the processor vendor. Figure 1.2 summarizes the modified organization of newer abstraction levels. As this system architecture optimizes the system performance utilizing characteristics of the user, applications and materials in a holistic manner, we term it as 'holistic architecture'.

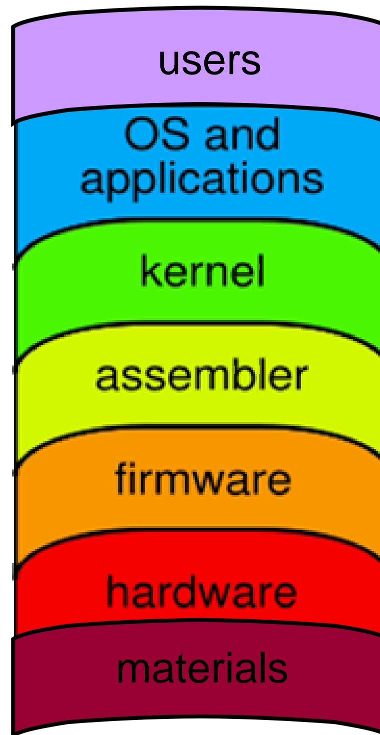


Figure 1.2. *Abstraction levels in holistic computer architecture.*

Note that, the primary objective of a computing system is to satisfy the user's expectations. Previous researchers have worked into user satisfaction while interacting with a system[17, 18]. But their objective was to dynamically optimize the operating system behavior to satisfy the user. We believe that including individual user's preferences to optimize system hardware utilization would lead to better performance and power. The results presented in this dissertation support this hypothesis. Additionally, analysis of the materials that lies at the lowermost end of the system spectrum opens up a number of opportunities to optimize the system performance. Every individual piece of hardware possesses unique properties even after going through the same manufacturing technology. Circuit designers typically consider the worst case scenario to predict the default voltage

properties of a processor chip. The hard constraint of reliability has created a gap between the default value and the threshold where a circuit can work flawlessly. We have shown that treating the correctness as an objective can improve the system performance with noted reductions in power consumption. The rest of the chapter summarizes major contribution of this dissertation.

1.2. Correctness-aware Application Adaptive Execution

We have looked into the trade-off analysis between reliability of a processor and its performance [19-21]. This research was aimed towards the development of a new programming model for network processors that would act as a bridge between the circuit designers and the computer architects. My work has questioned the traditional assumption about reliability and proposed an analysis which has been proven to effective in improving the system performance.

Traditionally, the circuit designers make sure of the fact that the designed chip should work at the worst case scenario. We have questioned this basic assumption about reliability. The reliability of the system has been compromised to gain in terms of performance. Please note that, while loosening the strict constraint on reliability, we have made sure that the system should not crash. We proposed the design and utilization of clumsy packet processors. We introduced a realistic model that determines the probability of a fault for a given cycle time of a cache and show that the delay of the cache and the energy consumed by the cache can be reduced significantly without incurring a large penalty on faulty behavior. Using simulation, we investigated an optimal point for trading off the reliability

for reducing cycle time of the data cache in a representative architecture. Moreover, a scheme is implemented to dynamically adjust the operation frequency of the data cache to achieve the desired objective (e.g., reduced energy).

1.3. Task Allocation based on statistical variation

We have proposed a task allocation scheme [22] that utilizes the probability distribution of the execution times of different modules in the networking applications. The task allocation scheme utilized the modular nature of networking applications. The goal for the research is to minimize the effects of execution time variation. Variation in execution time is an inherent property of processing. The proposed scheme can estimate this variation for different parts of the code and perform the task allocation accordingly. Results reveal several important characteristics of the proposed schemes. First, they show that the base task distribution scheme achieves high levels of scalability. In addition, the extended processing time and replication scheme help to improve the performance.

1.4. User-Experience Driven Optimizations

To explore the role played by the human factor in computer architectures, individual user's preferences over the system performance is analyzed during execution of different applications. A double blinded user study reveals that personal preferences vary greatly among users (and that a user's preferences vary dynamically during application run-time) [23, 24]. Existing Dynamic Voltage and Frequency Scaling (DVFS) techniques in high-performance processors select an operating point (CPU frequency and voltage) based on the utilization of the

processor. While this approach integrates OS-level control, such control is pessimistic about the user. Indeed, it ignores the user, assuming that CPU utilization is a sufficient proxy. A high CPU utilization leads to a high frequency and high voltage, regardless of the user's satisfaction or expectation of performance.

To remedy this limitation, we have developed User Driven Frequency Scaling (UDFS) that dynamically adapts CPU frequency based upon direct user feedback – as opposed to tracking CPU utilization, as is done by current methods. This dynamic power management scheme automatically adapts to different users and applications. UDFS effectively employs user feedback to customize processor frequency to the individual user. This typically leads to significant power savings compared to existing dynamic frequency schemes that rely only on CPU utilization as feedback. The amount of feedback from the user is reasonable, and declines quickly over time as an application or set of applications is used. Hence, it can reduce power consumption while still achieving high user satisfaction.

1.5. User-Perceived Performance Evaluation

Any architectural optimization (performance, power, reliability, security, etc.) ultimately aims to satisfy the user. The success of such an optimization relies upon the accuracy of its performance metrics as proxies for user satisfaction. Typically, such metrics are derived from low-level knowledge such as instruction throughput, hardware utilization, or operating system calls even though this knowledge is usually hidden from the user. We propose to derive these metrics not

from information that is “close to metal” and hidden from the user but rather with information that is “close to flesh” and apparent to the user. We describe and evaluate **PICSEL**, a dynamic voltage and frequency scaling (DVFS) technique that uses measurements of variations in the rate of change of a computer’s displayed screen to estimate user-perceived performance. The adaptive algorithms, one conservative and one aggressive, use these estimates to dramatically reduce operating frequencies and voltages for interactive applications while maintaining performance at a satisfactory level for the user. This is a collaborative project whose results have been shared by myself, and a fellow graduate student, Jack Cosgrove. My objective during this research was to explore the microarchitectural innovations involved in user-aware computing. Jack was primarily involved with the architecture of the display device of a system.

1.6. Process-aware Voltage Setting

Existing DVFS techniques are pessimistic about the CPU. They assume worst-case manufacturing process variation and operating temperature by basing their policies on loose worst-case bounds given by the processor manufacturer. However, as the manufacturing technologies are getting smaller, this conservative assumption becomes an important bottleneck. As transistors are reduced in size, it becomes harder to control variations in device parameters such as channel length, gate width, oxide thickness, and device threshold voltage. Therefore, the “one-size-fits-all” approach of current DVFS schemes is suboptimal in the presence of process variations.

We have developed a new power management technique, Process-Driven Voltage Scaling (PDVS). It creates a custom mapping from frequency and temperature to the minimum voltage needed for stability. It adapts to process variation, permitting processors to operate at their lowest stable voltages. This mapping is then used online to choose the operating voltage by taking into account the current operating temperature and frequency.

1.7. Dissertation Overview

The remainder of this dissertation is as follows. In CHAPTER 2, we present an analytical model that determines the probability of a fault in a circuit element for a given cycle time. Furthermore, we develop a framework to analyze and quantify the effect of hardware faults on networking applications. CHAPTER 3 discusses a novel clumsy processing environment where the hard reliability constraints are unleashed to gain in terms of system performance and power. The modular nature of networking applications and intelligent task allocation based on such properties are discussed in later part of CHAPTER 3. CHAPTER 4 demonstrates how direct user feedback can optimize a system's performance. Additionally we have shown how power management schemes can be benefited by customizing a CPU based on process characteristics. The evaluation of user-perceived performance and its utilization in a smart power management scheme is discussed in CHAPTER 5. Overall contributions of the dissertation are summarized in CHAPTER 6.

CHAPTER 2

Application-Level Error Measurements for Application Specific Processors

There is an inherent possibility of fault occurrence in any system. The sources for these faults can be different - they may arise from adverse environmental conditions [25], physical hardware defects, electronic noise, incorrect device utilization, or logical design flaws [26]. In addition, modern processors are advocating for aggressive scaling of the supply voltage (V_{dd}) and use smaller manufacturing technologies. This will increase the probability of fault occurrence. Increasing clock rate and the use of flip-chip packaging are expected to have adverse effects. Moreover, even if the probability of faults for a single transistor can be kept constant, the probability of faults in a processor will increase in parallel with the number of transistors on a chip. While it is critical to avoid these faults with careful circuit design and packaging, they can still occur and need to be addressed.

The effect an error has on a system is largely dependent on the hardware application. In most cases, omitting errors is not an option, i.e., the processor should be designed to capture and eliminate faults. This is the inherent nature of

the user expectation - a desktop processor or server is expected to work continuously for days or weeks without losing any data. In such cases, hardware faults are not acceptable. However, for other domains—such as networking and media applications—a certain level of error is acceptable, and the integrity of the system's behavior can be maintained despite potential faults. This is also related to the properties of the systems: networking software/systems are implemented with the assumption that the hardware can fail (e.g., routers can drop packets). Therefore, faults at a certain level are acceptable for such processors. In our work, we present a methodology to classify and measure the effect of hardware errors on networking applications.

Under the presence of faults, even if the system seems to be behaving correctly from outside, its operation may be affected. As a result, the system will operate differently depending on what kind of data becomes corrupted. For instance, in the presence of electronic noise, a single piece of transient data may get corrupted. This affects circuit behavior only momentarily. On the other hand, a static data element might be damaged—such as a lookup table, which is used for every packet operation that is processed by the system. This would affect the system for a longer period of time. Additionally, recovering from such errors is intuitively more difficult.

In this chapter, we highlight the need for application-level characterization of hardware faults. Particularly, we measure the susceptibility of Network Processors (NPs) to faults and their resulting behavior. Several applications from

the NetBench benchmarking suite [27] are studied and error metrics for each of these applications have been defined.

We start with building an analytical model that relates fault probability in a circuit element with the clock frequency. It is followed by a study where we introduce cache faults based on the analytical model and measure their effect on these applications. NetBench suite consists of a variety of applications that can be used to simulate a range of network processors' functions. Among these are routing, encryption, and packet filtering, all of which exhibit different behavior in the presence of faults.

We have examined and classified different kinds of errors that may occur in a network system. One type is marked as a volatile error, i.e., errors affecting data only temporarily. The other type is a nonvolatile error, i.e., errors affecting a static data structure.

We have analyzed the effects of errors on network applications. Our goal is to define data segments of these applications that can be used to measure their error behavior. Particularly, we study several networking applications and define error metrics for each of them. Then, we perform a study where we introduce cache faults and measure their effect on these applications. Specifically, our contributions in this dissertation are:

- We find a realistic model that determines the probability of a fault for a given cycle time of a cache and show that the delay of the cache and the

energy consumed by the cache can be reduced significantly without incurring a large penalty on faulty behavior,

- We classify errors in network applications depending on the extent of their effect on applications,
- We define several data structures to measure the extent of effect of errors in network applications,
- We simulate hardware faults and record the corresponding behavior of different applications.

2.1. Analytical Model of Error Probability

Injection of noise into a circuit node causes a signal deviation at that node. This signal deviation will affect the operation of the circuit or circuit block driven by the victim net. A functional failure is possible when induced noise is propagated and wrongly evaluated at the primary output. The parameters that determine if there will be a logic error are (i) the amplitude and the duration of the noise pulse, (ii) the type of the victim node and the circuit connected to the victim node, and (iii) the signal condition on the affected node. It is important to note that with increasing clock frequencies, a circuit node may suffer from reduced voltage swing, since there is not enough time to fully charge or discharge the load capacitance. C_{fs} in Figure 9 is the clock cycle time required to obtain the full voltage swing (V_{fs}) from zero to V_{dd}. Note that the supply voltage is kept constant at V_{dd}.

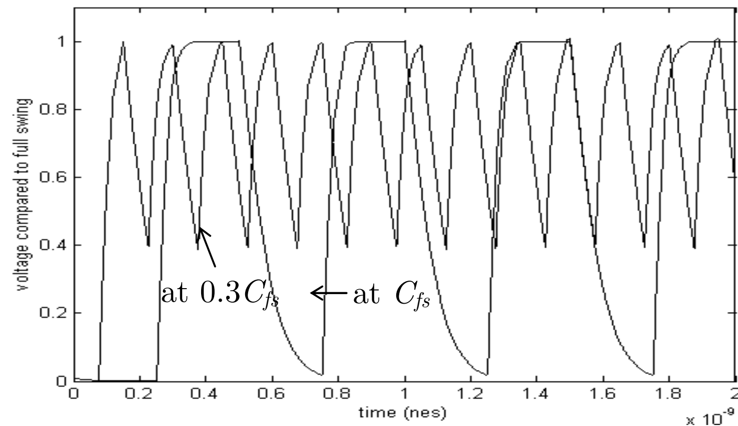


Figure 2.1. *Voltage at a circuit node at two different frequencies*

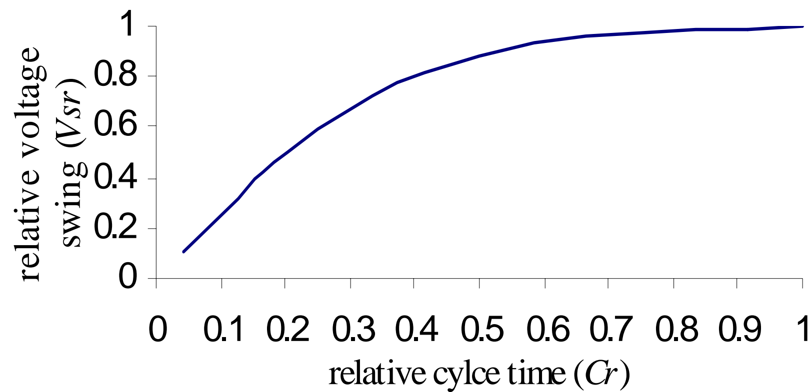


Figure 2.2. *Decrease of voltage swing with increase of frequency*

Figure 2.1 illustrates the decrease of voltage swing (Vs) with the decrease of clock cycle time (C). The clock cycle time and the voltage swing are normalized against the clock cycle at full swing (C_{fs}) and the full swing voltage (V_{fs}), respectively. The relative voltage swing is defined as $V_{sr} = V_s/V_{fs}$ and the relative cycle time $C_r = C/C_{fs}$. If the voltage swing changes, all the signals become faster by the same ratio independent of the capacitive load at a circuit node. Note that the change of voltage swing slows down at longer clock cycle time. This shape correctly maps the change of actual signals on-chip with time. Any signal at a

circuit node rises quickly at the beginning and as the signal reaches close to the full swing value it takes longer time for a certain change. The curve in Figure 2.2 has been produced by simulating a chain of gates driven by an inverter at different frequencies with constant supply voltage V_{dd}.

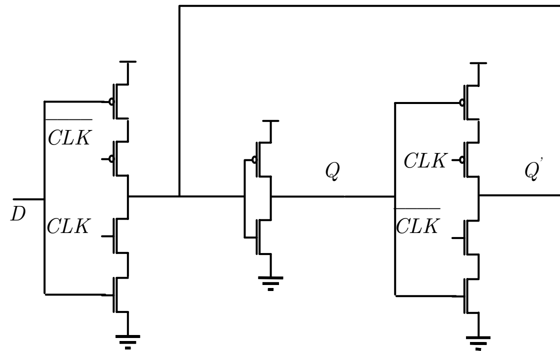


Figure 2.3. *A simple D Flip-Flop*

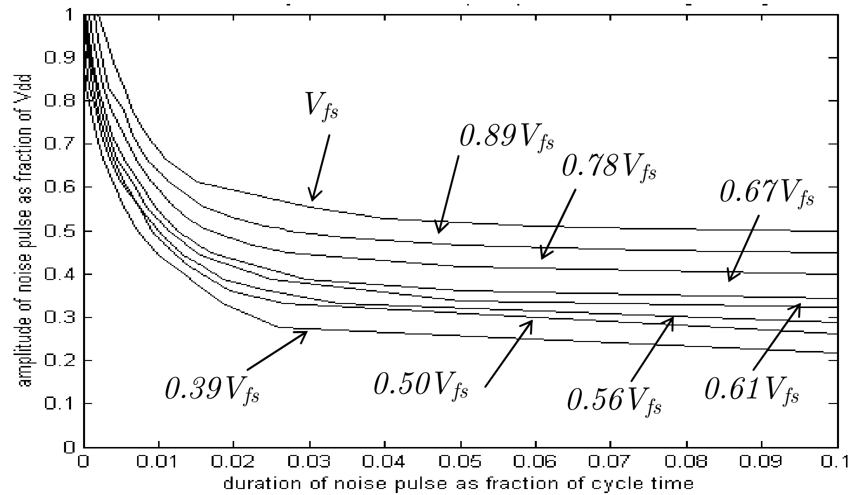


Figure 2.4. *Noise immunity curves of a D flip-flop at various voltage swings*

With a reduced signal level, a circuit node is more likely to suffer from logic failure due to a certain level of noise. Therefore, increasing frequency leads to higher probability of logic failure at a circuit node due to reduced voltage swing. The main advantage of static logic over dynamic logic is its robustness under the

influence of noise. But static logic may suffer from logic failure if there is a feedback loop. A static D flip-flop (as in Figure 2.3), which is common in registers, has a feedback loop that cannot recover from noise-induced errors. In these types of circuits there are three possible points where noise can be injected: the input, the clock and the feedback loop. The feedback loop is the most sensitive to noise. Even a small noise pulse on the feedback loop when the clock is falling or inactive will be propagated repeatedly through the loop and may ultimately destroy the logic information stored in the flip-flop. A set of noise immunity curves for the D flip-flop in Figure 2.3 is presented in Figure 2.4, which plots the relative noise duration (D_r) against the relative noise amplitude (A_r) at various voltage swings. Noise pulses of various amplitudes and durations have been injected into the feedback loop of a D flip-flop at different voltage swings, while keeping V_{dd} constant. SPICE simulations were used to determine the set of noise amplitudes and durations that cause a logic failure for different voltage swing levels. The area above each curve in Figure 12 represents the amplitudes and durations of a noise pulse that can cause logic failure. Hence, the lower the voltage swing the larger the area of noise amplitudes and durations that can cause an error. The relative noise amplitude is defined as $A_r = A/V_{fs}$, where A is the amplitude of the noise pulse, and the relative duration of noise $D_r = D/C_{fs}$, where D is the duration of the noise pulse. The highest curve is for the full voltage swing V_{fs} (swing from zero to V_{dd}). The lower curves illustrate noise immunity at voltage swings smaller than the full swing. It is important to note that the noise amplitudes and durations are not

equally probable. The probability of smaller noise amplitudes and noise durations are higher than larger amplitude pulses with longer duration.

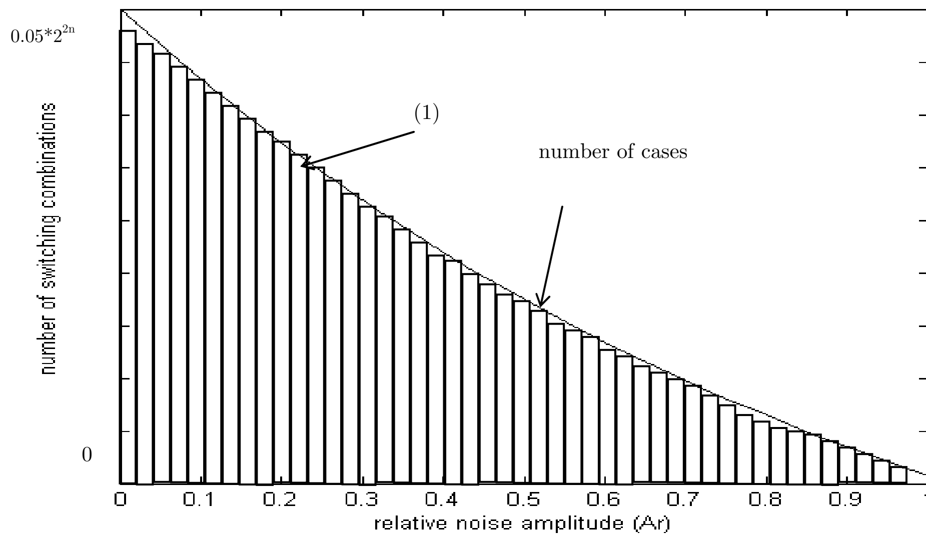


Figure 2.5. *Noise amplitude at various switching combination of neighboring lines of a victim line*

Consider a victim line, which has n neighbors significantly coupling to it. For noise injection into the victim line the total number of switching combinations of the neighboring lines is 2^{2n}. Only one switching combination results in the worst-case noise amplitude, which occurs when all the neighboring lines switch in the same direction. However, the number of cases where the effects of most of the neighboring lines cancel each other resulting in small amplitude of noise is large. We have found the number of switching cases between these two limiting cases, which result in a certain noise amplitude range. The results are plotted in Figure 2.5. This distribution can be approximated by an exponential as in (2.1).

$$\text{Number of cases} = K_1 e^{-K_2 A} \qquad (2.1)$$

The exact constants K_1 and K_2 depend on the number of lines (n) coupling to the victim line. For large n (greater than 16) this curve saturates to continuous probability distribution of the form

$$P(A_r) = 28.8 * e^{-28.8A_r} \quad \text{where} \quad 0 < A_r < \infty \tag{2.2}$$

$$P(D_r) = 10 \quad \text{for} \quad 0 < D_r < 0.1$$
$$P(D_r) = 0 \quad \text{for} \quad 0.1 \le D_r \tag{2.3}$$

The probability distribution of noise duration can be given by (2.3). The reason why D_r is uniformly distributed between 0 and 0.1 is that this is the range of rise time on chip as a ratio of the cycle time. Note that the noise duration is limited by these rise times, since noise occurs due to capacitive and/or inductive coupling of switching line to a victim line.

Once an aggressor signal settles, the noise pulse ends. Using equation (2.2) and (2.3), the probabilities (PE) of logic failure for a D flip-flop at different voltage swings have been obtained by the integration of the probabilities of noise pulse above each curve of Figure 2.6. Figure 2.6 plots the probabilities of logic failure against the relative voltage swings (V_{rs}). The probability number at full voltage swing are consistent with industrial and test data [28].

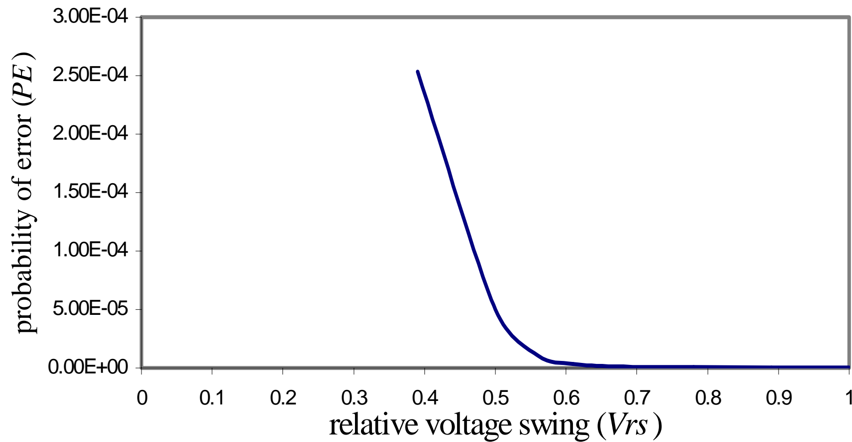


Figure 2.6. *Probability of error at different cycle time*

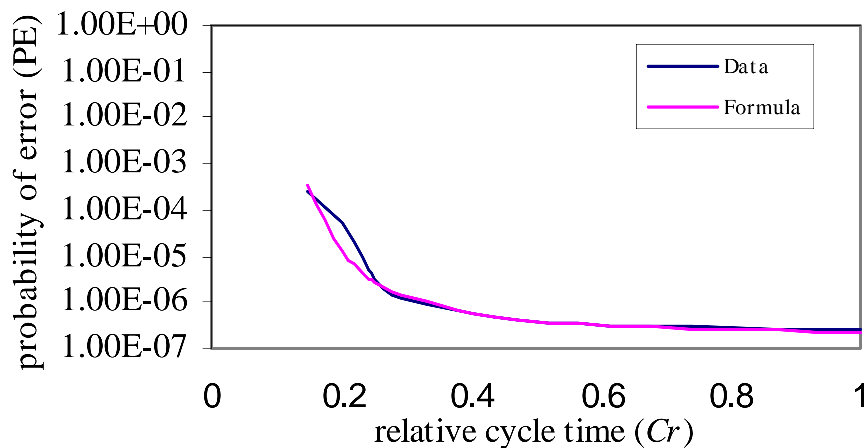


Figure 2.7. *Probability of error at various voltage swings*

The probability of error versus cycle time in Figure 2.7 has been obtained by the voltage swing variable from the two relations: cycle time versus voltage swing (Figure 2.2) and probability of error versus voltage swing (Figure 2.6). The relative cycle time Cr is always less than 1 for lower voltage swings. Similarly we can define relative frequency $F_r = f/f_{fs} = 1/C_r$, where f is the frequency and f_{fs} is the frequency at full voltage swing. PE is a single bit probability of error and is a

function of how fast a circuit is driven by allowing the voltage swing to decrease. The formula below shows the relation between PE and C_r and F_r.

$$P_E = 2*10^{-7}*e^{\frac{1}{6*C_r^2}} = 2*10^{-7}*e^{\frac{F_r^2}{6}} \qquad (2.4)$$

These formulae have been found by curve fitting for the data of the above curves. The curves in Figure 2.7, showing the data and the curve fitted formula, illustrate the accuracy of the formula. Note that if the circuit is pushed enough not to allow any voltage swing, the error probability will be 1. However, the circuit is never pushed to these limits. Note that, this particular fault model is applicable for a specific circuit element, register file in current work. The other parts of the circuit won't follow the same fault model. However, using similar procedure, it is possible to come up with accurate fault models for other parts of the processor. In our earlier studies we have developed a fault model which predicts the fault occurrence probability in the data cache [19].

The overclocking of the data cache can be implemented either statically or dynamically. For static implementation, the clock rate would be decided at the design time. This will be performed by setting the clock period higher than the estimated delay. This scheme won't require a separate clock for the register file. Dynamic implementation, on the other hand, would adjust the clock of the system to a higher (lower) value as the amount of error is below (above) a predetermined

threshold value. However, this dynamic adjustment has a high hardware overhead. Hence, in our work we utilize a static overclocking scheme.

2.2. Sources of Errors

An important trend driving microprocessor performance has been scaling of device sizes. Device scaling is the reduction in feature sizes and voltage levels of basic devices on the microprocessor. Aggressive scaling results in escalated power density and processor temperature, increasing the probability of faults [29].

Lowering the supply voltage in a microprocessor makes it more susceptible to noise. With a reduced signal level, a circuit node is more likely to suffer from logic failure due to a certain level of noise. Therefore, increasing frequency leads to higher probability of logic failure at a circuit node due to reduced voltage swing. In high frequency circuits, the analysis of errors due to logic failure has become an important research area [30].

Another consequence of the technology scaling is the smaller supply voltages and reduced capacitive values of the circuit nodes. This has raised reliability concerns due to the increased susceptibility to soft errors. Soft errors or transient errors are circuit errors caused by excess charge carriers induced primarily by external radiations, such as alpha particles and high energy neutrons [31-33]. While these errors cause an upset event, the circuit itself is not damaged. In different memory designs, these errors can cause a particular node to charge or

discharge and thus cause a bit flip. This is particularly true for SRAM cells used in caches.

Although designers aim to prevent all hardware faults while designing a system, faults can still occur due to external factors. Hence, fault tolerance has traditionally been an essential field of study and is becoming even more important for high-performance processors.

2.3. Error Classification

There are different types of error that may occur in a network system and their effects on the applications vary. An error can be classified as a *volatile error* if it affects the application in a local manner, e.g., an incorrectly received network packet or corruption of a temporary value. In general, these types of errors are likely to affect a limited amount of data and will not noticeably affect performance and/or application output provided that the error does not continually reoccur. While processing a networking application, we can pay less attention to volatile errors whose occurrence is limited to very few packets.

The other type of error is the *nonvolatile errors*, which affect the application more seriously. Such errors generally result due to the changes of static data structures, e.g., the routing table used in a NAT application. This type of error will have a permanent effect on the system. Since the data structures that are generated in the control plane are used for the processing of each packet, a fault during the execution of the control plane tasks may corrupt many

calculations over time before it is corrected. Hence, the effect of nonvolatile errors is likely to be more severe and measures should be taken to detect and prevent them.

It must be noted that the user makes an assumption about the completion of the application even if the fault probability is high. In reality, on the other hand, execution of an erroneous code diminishes the guarantee of completion. As the code may read erroneous data, it may turn into an infinite loop or may try to get access to some non-existent data during execution. Such events would cause the system to crash. This is a possible outcome for each of the applications we are investigating and is of interest to us for measuring the effects of faults. We classify such an error, which prevents the program from continuing its execution, as a fatal error. In case of a fatal error, the integrity of the systems is disrupted. Hence, we give special importance to fatal errors. In Section 2.7.3, we record the probability of a fatal error for different fault rates in the applications and report them separately.

2.4. Applications and Error Metrics

In this section, we discuss the networking applications studied in this project and present the data structures used to measure application errors for each benchmark program. We selected seven applications from the NetBench [27] suite. The applications are listed in Table 1. NetBench is a benchmarking suite designed for NPs. It contains applications representing level 3 tasks (e.g. route) as well as higher-level programs (e.g. MD5).

For each network application, important data structures and output of key function units are identified. Our goal is to make a comparison of these data values between the correct execution and a faulty execution. Using simulation results, we can calculate the statistical probability of an error to happen in the application. We can notice that a part of these data structures has more impact on the overall output than others (e.g. a routing table error is more important than an error in the ttl value calculation).

In this analysis, we simply list the structures that are important in the execution. These structures help us to scan the state of the application while it is executing. We have marked the Error Keys. We measure the effect of cache faults on these structures as discussed in Section 2.7.1.

In the following, we list the selected application followed by the application-level error metric used to measure the effects of faults.

CRC: The CRC-32 checksum calculates a checksum based on a cyclic redundancy check as described in ISO 3309 [34]. CRC-32 is used in Ethernet and ATM Adaptation Layer 5 (AAL-5) checksum calculation. The code is available in the public domain [35]. The errors are measured using two data structures: the crc table and the crc accumulator value calculated for each packet. Note that the errors in the crc table are more important as they would affect multiple packets during the processing. Any error on the accumulator calculation part would concern only one packet.

TL: TL is the table lookup routine common to all routing processes. We have used radix-tree routing table, which was used in several UNIX systems. The code segment is from the FreeBSD operating system [36]. The error metrics in the TL application are: the radix tree nodes traversed and the RouteTable entry for each packet.

Table 2-A. *NetBench Applications and Their Properties*

Application	***Arguments***	***No. of inst. simulated [M]***	***No. of cache access [M]***
CRC	5000	145.8	59.8
TL	128 5000	6.9	3.9
ROUTE	128 5000	14.2	7.1
DRR	128 5000	12.9	7.9
NAT	128 5000	11.4	5.6
MD5	5000	209.1	73.2
URL	small_inputs 5000	497.0	249.1

ROUTE: IPv4 routing according to RFC 1812 [37] is implemented in the Route application. When a packet arrives in a router its next network hop is decided by the router. Route implements the table lookup along with internet checksum (for the header). During processing, there are changes in the header (for example, the Time-To-Live value). It may fragment the packet and forward it. The code is also from the FreeBSD operating system [36]. The values observed in the ROUTE application are: the entries in the created RouteTable, the checksum value, the ttl value, and the radix tree entries traversed for each packet.

DRR: Deficit-round robin (DRR) scheduling [38] is a scheduling method implemented in modern network switches. In DRR, all the connections through the

router have separate queues. Using these queues, the router tries to accomplish a fair scheduling by allowing the same amount of data to be passed from each queue. The implementation is based on the algorithm by Shreedhar and Varghese [38]. The data values in the DRR application are: the entries in the created RouteTable, the radix tree entries traversed for each packet, the value of the deficit list for each packet, and the deficit information read for the packet.

NAT: Network Address Translation (NAT) is a common method for IP address management. NAT operates on a router, usually connecting two networks, and translates the private (not globally unique) addresses in the internal network into legal addresses before packets are forwarded onto the public network. Hence, for any departing packet, the source IP on the packet should be changed. Similarly, the destination address on any incoming packet should also be modified. The program accomplishing this task is using several routines from the FreeBSD operating system [36]. The data values used for measuring errors in NAT are: initial IP source address, value in the interface for translation, translated IP source address, the IP destination address after translation, the entries in the NAT table, and the radix tree entries traversed for each packet.

MD5: Message Digest algorithm (MD5) creates a signature for each outgoing packet, which is checked at the destination [39]. The signature is cryptographically secure, hence if the received packet does not match the signature, then the receiver will assume that the packet is unreliable and discard it. The implementation is from RSA Data Security, Inc. [40]. The errors in MD5

are binary errors. So they are easily detectable. In other words, if the output string of the erroneous execution does not exactly match the correct execution, the packet is said to be processed incorrectly. We then measure the fraction of packets incorrectly processed.

URL: URL implements URL-based destination switching, which is a commonly used content-based load balancing mechanism. In URL-based switching, all the incoming packets to a switch are parsed and forwarded according to URL. For example, all image requests might be sent to an image server. This application increases the utility of specialized servers in a server farm. The implementation is based on the description from PMC-Sierra [41]. The data structures in the URL application that are observed are: URL table entries, final IP destination address, RouteTable entries, the checksum value, the ttl value, and the radix tree entries traversed for each packet.

2.5. Error Injection and Measurement

We introduced faults in the applications by simulating random faults in the data cache. Erroneous values are inserted randomly in the register files and propagated during the execution of the applications. At the same time, the proposed error metrics for each of the applications are scanned. Every mismatch of the values between the correct simulation and the erroneous simulation is counted as an application error for the corresponding application metric. We observe some error metrics reacting more sensitively to hardware faults than others. As discussed in Section 4.3, the error metrics could be classified as volatile and non-volatile

errors. For example, any error in the RouteTable entries is permanent and affects the system severely. On the other hand, an error in the ttl value of a packet during routing is limited to the corresponding packet, hence is a volatile error.

We assume a processor architecture similar to a generic Network Processor (NP). We model a relatively simple execution core with a local instruction cache, a local data cache, and a shared cache that corresponds to a level 2 cache as described in Section 2.6. Although we apply our ideas to a packet processor, they can be applied to any type of processor that executes applications with fault tolerance (e.g. media processors). Therefore, we selected more generic programmable processor architecture.

One important aspect of the cache accesses is whether to include a fault detection scheme or not. In this work, we have assumed an architecture where no fault detection scheme (e.g., parity) is employed. In addition, we assume that the data in the level 2 cache (or the next level of memory hierarchy) will always be correct if it is not written back from the first level cache. So, if a fault is detected, we can access the data from the level 2 cache. Therefore, error correction techniques (such as Hamming codes) would incur unnecessary complication on the design and energy consumption and hence are not considered in our studies.

2.5.1 Fallibility Factor

We need to introduce a measurement index to analyze the effect of hardware errors on the networking applications. Since the processor is going to

make errors, traditional approaches such as delay, energy, or energy-delay product would be insufficient. We define the metric fallibility as the probability of the processor making an error for the application. One can use the number of hardware faults that are not detected to measure the fallibility. However, due to the application-specific nature of our target architectures, we use application errors in the fallibility factor as discussed in Section 2.5.1. Particularly, fallibility corresponds to the fraction of packets that have any type of errors. Note that even if the packet is correctly forwarded, it can still contribute to the fallibility rate. For example, if the ttl value of the packet is different than what it would be in case of correct execution, we consider the packet to have an error.

2.5.2 Fatal Error Probability

We pay special attention to fatal errors. Since fatal errors prevent other packets from being processed[1], we calculate the number of packets successfully processed until the occurrence of a fatal error. The reported fallibility factors are based on this number. We also report the probability of a fatal error in addition to the fallibility factor. Particularly, we record the probability of a fatal error with increasing error introduction rate. Increased hardware errors make the system more susceptible to termination. As a result fewer packets can be processed successfully at higher error introduction rate.

---

[1] Majority the fatal errors we have observed during our simulations are caused by the execution getting into an infinite loop.

The cause of the fatal errors can be attributed to several factors and we can classify the errors accordingly. A fatal error may occur due to an unimplemented system call or an access to restricted or non-existent memory location. The system crashes if one of these fatal errors occurs. However, the system may run into an infinite loop because of an error. We classify this fatal error as the "silent error".

We propose different remedies for the fatal errors depending on their nature. The destructive errors can be taken care of by higher levels of the system (e.g. operating system) that reset the system to a stable state to prevent a system crash. However, to prevent the "silent error", we can implement the check-pointing scheme to prevent the system from running into an infinite loop.

2.6. Simulation Environment

We use the SimpleScalar/ARM [42] for our simulations. We modified the input set to model a processor similar to execution cores in a variety of Network Processor architectures. Particularly, we simulate a processor similar to StrongARM 110 with 4 KB, direct-mapped L1 data and instruction caches with 32-byte line-size, and a 128 KB, 4-way set-associative unified L2 cache with a 128-byte line-size. We modified the applications to output the values of data structures mentioned in the previous section. As discussed in 2.4, the data metrics can be divided into two major categories. Some of them record the control structure of the application and the rest describe the packet processing tasks of the networking applications. All the applications dump the corresponding data metrics into a file that is processed later to calculate the application errors. Since there is always a

probability of early termination due to fatal errors, we also record the total number of packets processed in each simulation.

The simulator is modified to introduce random errors into the execution and to simulate the effects of the introduced errors. The architecture has been designed to propagate the introduced errors in subsequent stages. However, we must remember that it is the inherent nature of the applications, which enables them to limit the effect of the volatile errors within a particular packet. We chose an initial error probability of $12*10^{-8}$ per bit, as reported by Shivakumar et al. [30]. The error rate is calculated for each bit accessed independently. Therefore, we do not simulate the effect of relation between errors. Then, we increase the error rate in steps until it is set to $819.2*10^{-8}$.

2.7. Simulation Results

2.7.1 Application Error Measurement

This section describes the simulation results observed for selected NetBench applications. For different error rates, effects on the data structure of each application discussed in Section 2.4 are recorded. Figure 2.8 to Figure 2.14 describe the behavior of seven selected network applications for different error introduction rate. Figure 2.8 presents the results for the ROUTE application. Intuitively, the faults in the static data structures (volatile error) should have significantly more impact on the application behavior. This can be observed for initialization error. However, for most error types, the difference is not drastic. This behavior is due to the shorter length of the static data structure initialization and modification of the

application. In most of them, we spend less time in the control structures. For networking applications, majority of time is spent on processing the packets. Therefore, although each fault happening during initialization has larger impact on the error rate compared to the faults during packet processing tasks, the overall impact of errors during the static data structure tasks is not drastically more on the application errors. This is an encouraging result, because in many cases the processor will not have information about the type of task it is executing. Since the effects of control plane tasks are relatively smaller, the overall impact of errors can be kept minimal.

Figure 2.11 presents the results for the NAT application. Similar trends can be observed for this application as well. Particularly for the NAT application, we see that errors due to faults during packet processing tasks have more impact on the application behavior than the faults during static data structure tasks. This can be attributed to the fact that the NAT application does a lot of processing over each packet. As a result, the probability of an error during data processing tasks increases.

The results for the remainder of the applications are not discussed in detail due to their similarities with the presented results. However, all of them show similar characteristics of the applications under erroneous execution. In general, most applications can easily tolerate a small probability of error.

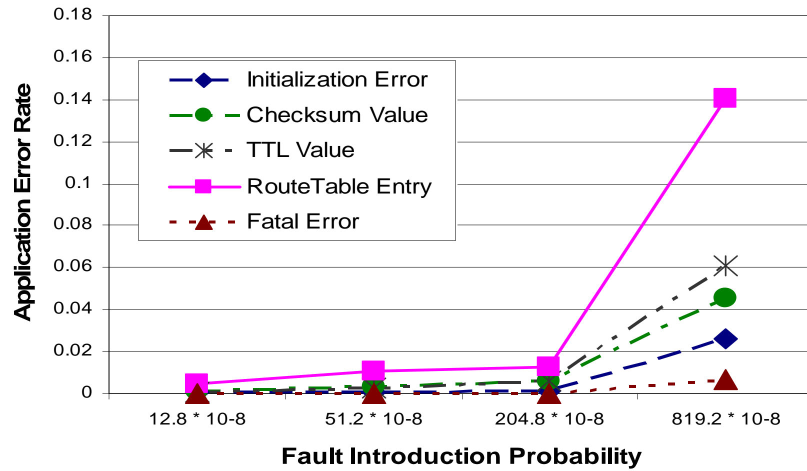


Figure 2.8. *Error Generation probability for Route application*

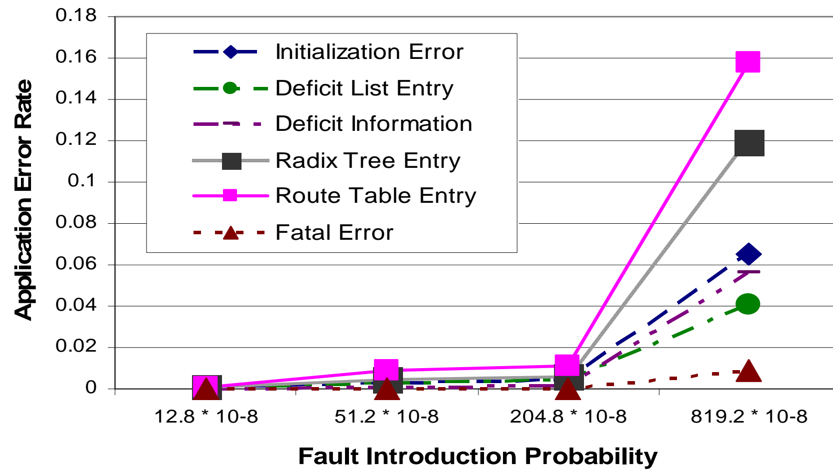


Figure 2.9. *Error Generation probability for DRR application*

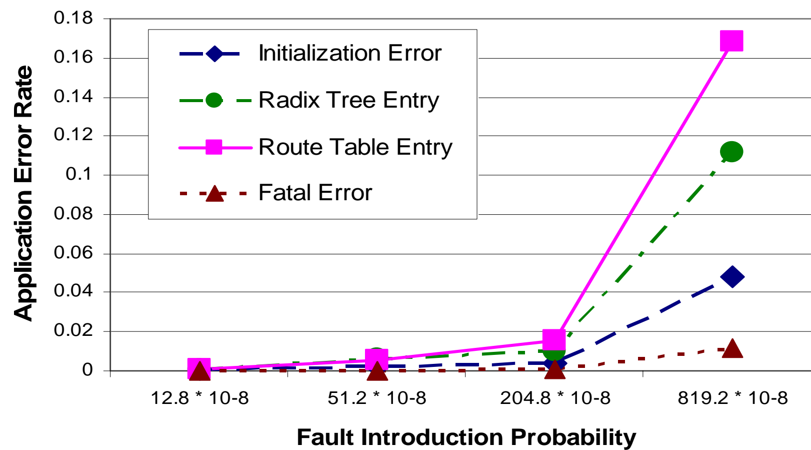


Figure 2.10. *Error Generation probability for TL application*

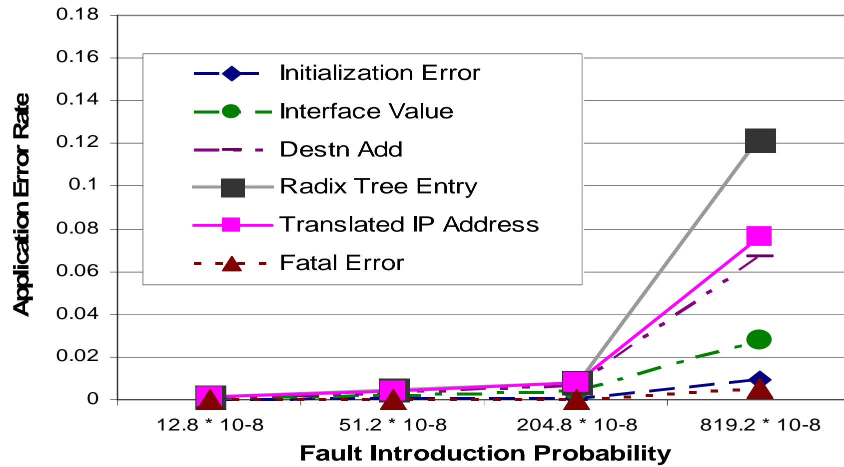


Figure 2.11. *Error Generation probability for NAT application*

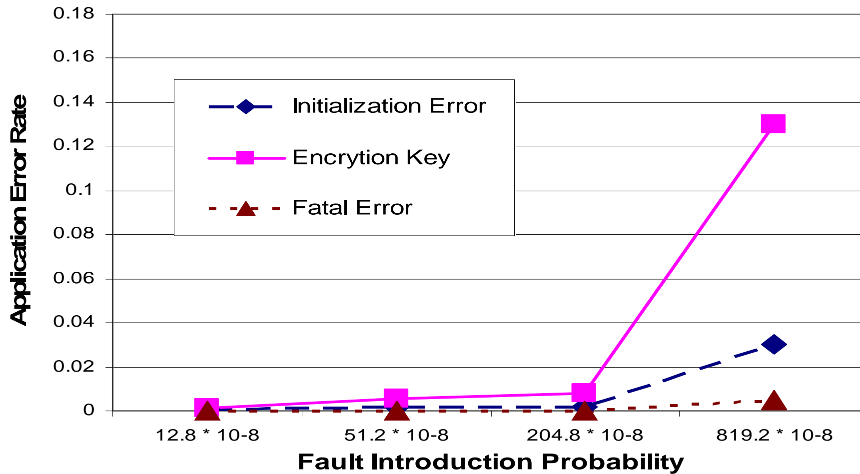


Figure 2.12. *Error Generation probability for MD5 application*

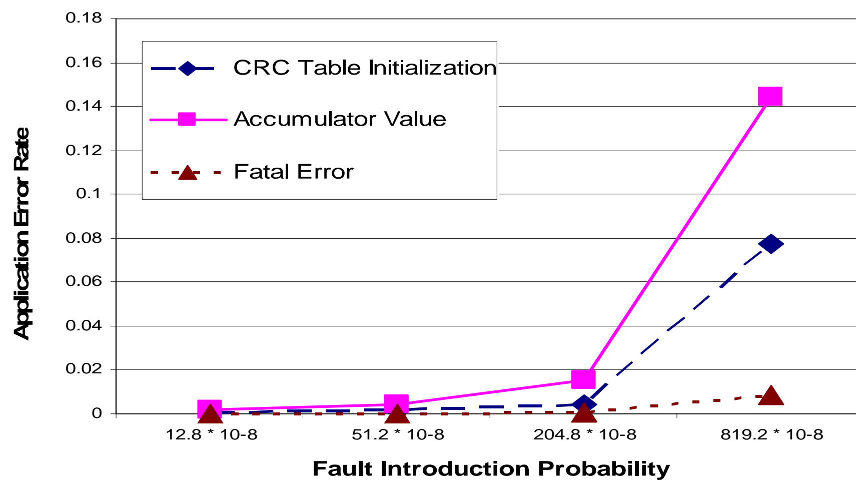


Figure 2.13. *Error Generation probability for CRC application*

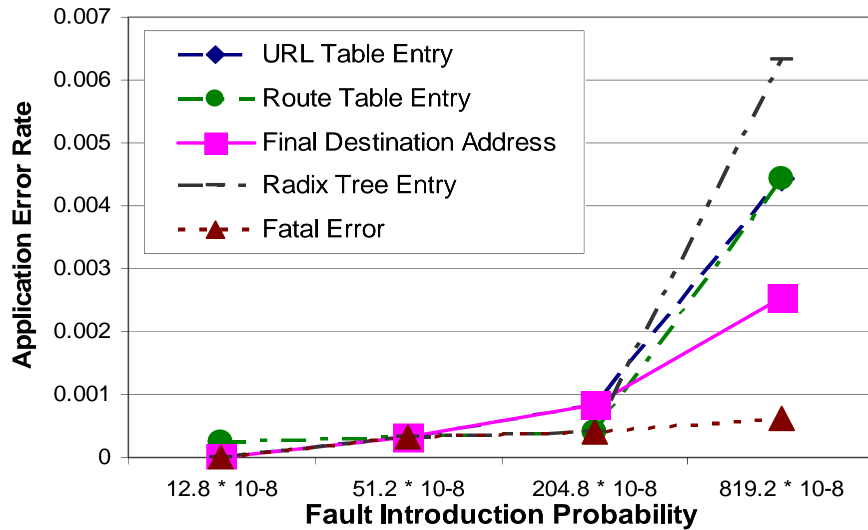


Figure 2.14. *Error Generation probability for URL application*

With the increase in error introduction rates, the applications start producing erroneous outputs. This is expected as increased hardware errors will definitely perturb the system integrity. The extent of application errors depends on the nature of the application and the error metric being observed. Some of the application metrics (e.g. ROUTE table entry) are affected easily, although being volatile errors, the effect of the erroneous behavior is limited to a single packet. We see that the fault introduction rate in the data cache can be increased up to 4 times without causing a major impact on the application output. During the simulations, we have seen that not all the faults have an impact on the application output. On average we have only observed an error for approximately 15% of the faults.

2.7.2 Fallibility Factor

In almost all the applications, we see that increasing the error introduction rate increases the fallibility factor. For lower fault introduction rate, the fallibility

factor suggests that the applications are not affected by the hardware faults. However, with increase in the error introduction probability, the integrity of the system becomes imbalanced which is properly reflected in the fallibility factor values. We recorded the probability of different application errors for each of the seven NetBench applications. The fallibility factor is obtained by adding 1 with the sum of all application error probabilities for each application.

Table 2-B. *Fallibility Factor of Different Applications*

Appln.	**Fallibility Factor**			
	$12.8*10^{-8}$	**$51.2*10^{-8}$**	**$204.8*10^{-8}$**	**$819.2*10^{-8}$**
CRC	1.0023	1.0038	1.0073	1.0524
Tl	1.0010	1.0063	1.0159	1.1350
ROUTE	1.0003	1.0008	1.0013	1.0175
DRR	1.0000	1.0010	1.0023	1.0076
NAT	1.0003	1.0020	1.0035	1.0770
MD5	1.0000	1.0115	1.0552	1.2610
URL	1.0003	1.0013	1.0025	1.0177

Table 2-B gives us a common framework to compare the behavior of the network applications subjected under hardware errors. The MD5 application shows maximum sensitivity towards errors. It has a fallibility factor of 1.261 when the error introduction rate is highest ($819.2*10^{-8}$). This rate means that on average in 26.1% of the packets an error key differs from the execution without any faults.

2.7.3 Fatal Error Probability

Each application can sustain the effect of the introduced error to varying extent. For smaller error rates we observed the execution of the application without any observable error in the data structures and the application output.

With increase in the error introduction rate, the applications started to produce erroneous outputs and data structure values. For larger error rates, the overall data integrity of the system was lost and the applications crashed. This is indicated by the fatal error probability. Figure 2.15 shows the probability of fatal errors at different error introduction probabilities. The figure suggests that we should never allow the system to work in an environment with high error introduction probability. This would cause a fatal error. A network system with errors in few packets is acceptable. However, an unstable system is certainly not desirable.

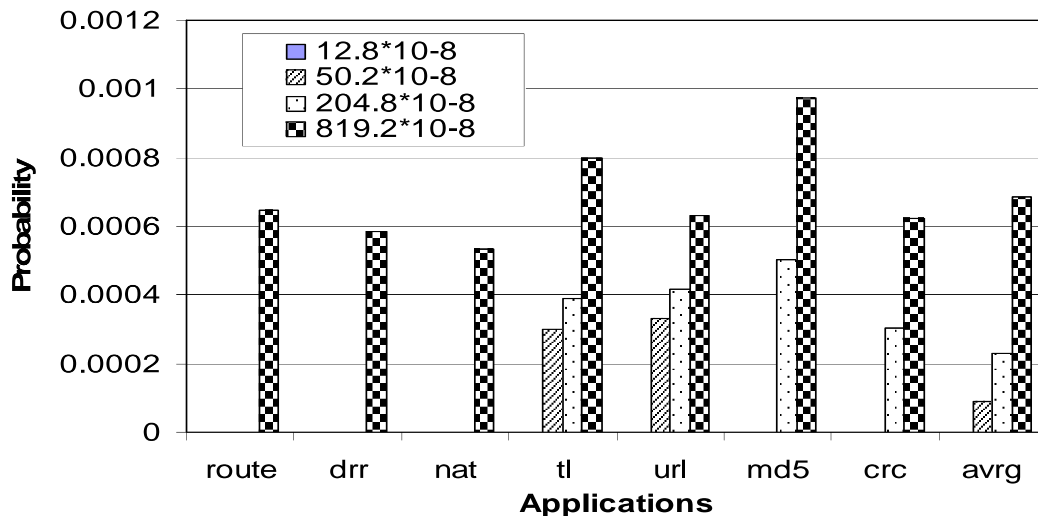


Figure 2.15. *Fatal Error probability for different applications*

2.8. Related Work

High transient-error tolerant system design has traditionally been considered in the context of systems that operate in high-radiation environments or in outer space, where there is a heavy concentration of alpha-particles and

atmospheric neutrons [43]. Recent IBM Research showed that computer systems are susceptible to transient faults induced by these particles [25]. In the circuit verification area, there has been a strong emphasis on reliability which is an important problem in IC fabrication. Earlier researches have studied potential errors in the pre-silicon [44] stage. Additionally errors subsequent to the fabrication process [45] have been analysed. High transient fault resilient computer systems design [46] has gained greater significance due to the combined effect of higher integration densities, lower voltages, and faster clock frequencies.

Fault injection is an attractive method for validation for estimating the dependability of computer systems [47]. Studies have already shown that the workload has a significant effect on the dependability measures [48, 49]. However, in our case we have investigated the application-level behavior of networking programs under hardware faults.

CHAPTER 3

Reliability-Performance Tradeoff Analysis

Architectural optimization based on application specific characteristics is one of the emerging trends in high performance computing. Traditionally, computer architects utilize application characteristics to explore higher levels of Instruction Level Optimization (ILP). They systematically evaluate application performance improvements associated with architectural enhancements that embodies acceptable cost/performance tradeoffs while reducing stalls in the microarchitecture. In this chapter, we have utilized novel system attributes to enhance system performance in modern microprocessor system.

First we will discuss how we can tradeoff reliability to gain in terms of performance in a computing system. With technology scaling, variability in transistor performance is increasing continually making them more prone to fault. On the other hand, Moore's law will enable us to billions of transistor in a single die. Figure 3.1 [50] depicts the trend starting from 2001 with 130nm technology generation, with a 300mm^2 die capable of integrating one billion transistors. The curve shows by 2015 we will have 100B transistors on a 300mm^2 die, with almost 1.5B transistors available for logic. The logic transistors tend to be larger than

transistors in the memory, take larger space, and consume more power. With an abundance of transistors that may not work reliably all the time, it is possible to design a reliable architecture with unreliable circuit elements. This kind of optimization would be most useful more application domains such as networking, media processing where an inherent robustness is present in the system.

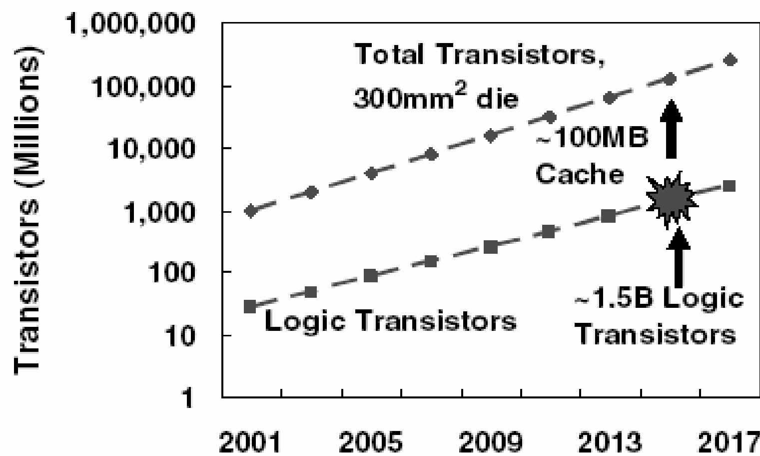


Figure 3.1. *Transistor Integration Capacity*

For networking application domain, we have observed a large amount of modularity in applications. With the emerging trend of using multiple cores in network processing systems, task allocation is an important bottleneck for performance optimization. We propose an intelligent task allocation scheme that utilizes modularity in networking application and statistical information about module processing. Overall, this chapter summarizes how holistic architecture framework can improve system performance using the following system attributes:

- Inherent system robustness
- Variability in application module processing

A. Clumsy Processing in Packet Processors

Hardware faults can occur in any computer system. Although faults cannot be tolerated for most systems (e.g., servers or desktop processors), many applications (e.g., networking applications) provide robustness in software. However, processors do not utilize this resiliency, i.e., regardless of the application at hand, a processor is expected to operate completely fault-free. In this chapter, we question this traditional approach of complete correctness and investigate possible performance and energy optimizations when this correctness constraint is released. We first develop a realistic model that estimates the change in the fault rates according to the clock frequency of the cache. Then, we present a scheme that dynamically adjusts the clock frequency of the data caches to achieve the desired optimization goal, e.g., reduced energy or reduced access latency. Finally, we present simulation results investigating the optimal operation frequency of the data caches, where reliability is compromised in exchange of reduced energy and increased performance.

3.1. Introduction

Over the last decade, in spite of the complexities of new manufacturing technologies and increasingly complicated architectures, designers have been able to steadily push the limits of performance of microprocessors. This is achieved through optimizations at the architectural level (such as aggressive pipelining strategies) and at the circuit level (such as smaller feature sizes). As we move into

deeper sub-micron technologies, the complexity of pushing the circuit performance has become an important obstacle. Increased heat dissipation and sub-micron effects are two examples of the limitations on the optimizations at the circuit level. In this work, we design a micro-architectural optimization to aid the circuit designers to overcome such hurdles. Particularly, we will allow the clock frequency of the data cache to go beyond the specifications of the circuit designer. Instead of performing this "over-clocking" uninformed, we will first explore the relation between the operating frequency (i.e., clock frequency) of a cache structure and its robustness. As we increase the clock frequency, the probability of a fault in the data cache accesses increases. This may result an erroneous execution of the applications. Hence, we name our proposed architecture a clumsy packet processor. In our approach, we first develop a model for estimating the hardware faults when the clock frequency is changed. This model will allow us to develop ultra-low power cache structures. In addition, the delay of the components will also be reduced. The disadvantage of this optimization is that the probability of hardware failure reduces the reliability of the processor. Overall, our goal is to investigate the trade-offs at the application-level, architecture-level, and circuits simultaneously in the context of packet processors. We use the term packet processor for any type of processor handling packets in a networking hardware. These range from network processors (NPs) to ASICs and general-purpose microprocessors used in networking hardware.

In all computer processors there is an inherent possibility of faults[2] being introduced into the system. These faults may arise from any of several sources such as adverse environmental conditions [51], physical hardware defects, electronic noise or logical design flaws [52]. Moreover, this fault problem is expected to be even more pressing in the future due to aggressive scaling-down of the supply voltages (V_{dd}), increasing clock rates, and the use of flip-chip packaging. While it is critical to put every effort to avoid these faults by careful circuit design and packaging, they can still occur and need to be addressed. Hence, we should consider reliability trade-offs even during the design of the processors, which will operate completely under the specified conditions.

The effect a fault has on a system is largely dependent on the application in question. In most cases, omitting faults is not an option, i.e., the processor should be designed to capture and eliminate faults. This is the inherent nature of the user expectation. However, for other domains—such as networking and media applications—a certain level of error is acceptable, and the integrity of the system's behavior can be maintained despite potential faults. This is also related to the properties of the systems: networking software/systems are implemented with the assumption that the hardware can fail (e.g., ROUTErs can drop packets).

Regardless of a fault's source, the system will operate differently depending on the corrupted data. Electronic noise may lead to the corruption of a single piece

---

[2] A fault is an incorrect execution of the hardware. An error is defined to be an incorrect outcome of an application due to a fault.

of transient data and affect behavior only momentarily. On the other hand, a static data element might be damaged—such as a lookup table—disrupting the system for a longer period of time and perhaps making recovery from the error more difficult. In this research, we analyze the susceptibility of a data cache to faults and the resulting behavior for packet processors. Particularly, we study several networking applications and define error metrics for each of these applications. We first make the distinction between the control plane and data plane tasks in these applications and measure the error behavior of the applications under different operation frequencies in these segments. Then, we perform a study where we introduce cache faults and measure their effect on these applications. Our goal is to extract optimal execution properties of the caches for different applications. We also present a scheme that dynamically adjusts the processor properties to achieve reduced energy consumption and/or increased performance. Specifically, our contributions in this chapter are:

- We propose the design and utilization of clumsy packet processors,
- We discuss simulation results investigating an optimal point for trading off the reliability for reducing cycle time of the data cache in a representative architecture,
- We implement a scheme to dynamically adjust the operation frequency of the data cache to achieve the desired objective (e.g., reduced energy).

There is also an increasing motivation to utilize NPs in wireless systems. In such systems, energy consumption is arguably the most important design criteria. Our optimization scheme reduces the execution delay and the energy consumption simultaneously.

The types of errors examined are similar to those in previous chapter (Section 2.3 and 2.4). One type is considered to be a *volatile error*, affecting data only temporarily. In general this type of error will only concern a limited amount of data, and will not noticeably affect performance provided that the error does not continually reoccur. The other type is a *nonvolatile error*, which has an effect on a static data structure (e.g., the routing table). This type of error will have a lasting effect on the system. Our goal in this chapter is to define data structures in these applications that can be used to measure their error behavior.

3.2. Applications and Error Measurement

In this section, we discuss the networking applications studied in this project and present the error metrics used for each application. We selected seven applications from the NetBench [27] suite. The applications are listed in Table Table 2-A. NetBench is a benchmarking suite designed for NPs. It contains applications representing level 3 tasks (e.g., ROUTE) as well as higher-level programs.

As a metric of "reliability", we first identify important data structures and outputs of key function units for each application. Our goal is to make a

comparison of these data values between the correct execution and an execution with faults (Section 3.3). Thereby, we will measure the probability of an error in the application. Some of these data structures have more impact on the overall output than others (e.g., a routing table error is more important than an error in the ttl value calculation). However, in this study we do not assign weights to them. Note that this type of measurement assumes that the application executes to completion even under faults. However, we are executing erroneous code (i.e., a code that will read erroneous data). As the data values are changed, it is possible that the application might fall into an infinite loop or even cause the system to crash. This is of interest to us for measuring the effects of faults. Therefore, an error, which prevents a complete execution is a special one called a fatal error.

3.3. Clock Variation and Fault Detection

We assume a processor architecture similar to a generic Network Processor (NP). We model a relatively simple execution core with a local instruction cache, a local data cache, and a shared level-2 cache. Although we apply our ideas to a packet processor, they can be applied to any type of processor that executes applications with fault resiliency (e.g., media processors).

One important aspect of the cache accesses is whether to include a fault detection scheme or not. If we don't provide the processor with an error detection and correction scheme, there is a possibility of the system to crash because of the occurrence of a fatal error. Moreover, as we are trading off reliability for performance and power of the system, it is a good idea to detect and correct faults

for the system to perform without any noticeable problem. There is a large space of possible implementations for error correction. Our framework can utilize any of these techniques. However, these techniques (such as Reed-Solomon or Hamming codes [53]) are usually computationally complex. Hence they would incur a performance overhead. Moreover, it could add extra cost due to additional hardware logic. We used simple parity checking to detect faults in a cache block. Upon detection, we have defined simple, cost-effective error correction schemes as discussed in the following sections.

In Section 3.4, we will experiment with a processor architecture where cache blocks are protected with parity and a processor architecture without any fault detection scheme. We are modifying the clock frequency of the level-1 cache only. Hence, we assume that the data in the level-2 cache will be correct unless an incorrect value from level-1 is written to it. Therefore, if a fault is detected, we can access the data from the level 2 cache. As the error correction techniques (such as Hamming codes) would incur unnecessary complication on the design and energy consumption, they are not considered in our studies.

Once a fault is detected, we have different options of recovery. A fault might be caused during the read in which case the actual data in the cache is actually correct or during the write to the cache. We cannot determine the exact source of the fault. The first technique we utilize assumes that every fault observed

is a write fault. Therefore, for every fault detected, it invalidates the cache block[3] and starts accessing the level 2 cache. This strategy is called a ***one-strike*** strategy. The second strategy accesses the cache after a fault and if another fault is detected, it invalidates the cache block and accesses the level 2 cache. This strategy is called a ***two-strike*** strategy. Similarly, a ***three-strike*** strategy accesses the level 1 cache twice before invalidating the block. Even if the processor employs a fault detection mechanism, there is still a chance of faults. Therefore, the application can behave erroneously.

Over-clocking the cache can be utilized during the design process of a processor. However, this is hard to achieve for programmable processors (such as Network Processors), because different applications might require different levels of reliability. Therefore, in the next section we also present results for a ***dynamic frequency adaptation*** technique. In this scheme, the processor adapts the operation frequency of the data cache according to the faults it has observed. Particularly, it records the number of parity failures during execution epochs. For our simulations, after the completion of the processing of 100 packets, the processor makes a decision for whether to increase the frequency, to keep it in its current state, or to decrease it depending on the number of faults. Note that the possible frequency settings are discrete. Hence, when the frequency is changed, it will be set to the next frequency level available. Whenever a frequency change is

---

[3] If the cache has sub-blocks, only the corresponding portions of the cache block can be invalidated and accessed from the level 2 cache. However, in this research we do not study such cache structures.

made, the number of faults in the previous epoch is stored. During the decision, if the number of faults is more than X_1% of the last stored fault rate, the frequency is reduced. If the fault rate is less than X_2% of the last stored rate, the frequency is increased. For all other rates, the frequency is not changed. A detailed study reveals that setting X_1 to 200% and X_2 to 80% overall results in the best performance of the dynamic scheme. This also relates to the fault model we have developed in Section 5. As shown in Figure 5, the clock cycle can be reduced by almost 60% before we observe a major increase in the number of faults. Depending on the packet processing time, the X_1 and X_2 values will lean towards increasing the frequency until a significant increase in the number of faults.

Most networking applications have application errors proportional to the number of faults occurred during the processing of a packet. The dynamic frequency adaptation technique observes the packet processing and makes the decisions for a constant number of packets (instead of time). This allows the system to dynamically adjust to the properties of the application. This information is usually available to the cores.

3.3.1 Implementation of the Cache Overclocking Architecture

Overclocking is applied to the L-1 data cache only, so we need to synchronize the cache with the rest of the core. For static over-clocking, this is straightforward. If the original data cache latency is 2 processor cycles and the cache is over-clocked by 50% or more, the cache latency will always be 1 processor cycle and the processor will be designed accordingly. In addition, in the static case,

we do not even need a separate clock signal to the data cache. The clock input of the cache can be multiplied to 2 clock cycles from a single one.

The incorporation of the dynamic overclocking is more complicated. In this case, we need a separate clock signal to the cache, and more importantly, we need to be able to adjust between a pipeline with 2-cycle cache latency and 1-cycle cache latency. Note that, we don't change the frequency of the data cache frequently. At the completion of processing for every hundredth packet, a decision is taken about the changing the cache clock frequency. In addition, note that dynamically varying the clock frequency of the cache is easier to implement than varying the supply voltage [54]. This can be achieved while the cache is being accessed and there is no need to flush the cache. In accordance with this, we incur a 10-cycle penalty whenever the frequency is dynamically varied. In addition, the hardware to implement variable clock rate is also quite simple. We assumed that the frequency can be increased by 50%, 100%, or 300%, corresponding to C_r values of 0.75, 0.5, and 0.25.

3.3.2 Error Injection and Measurement

We introduced faults in the applications by simulating random faults in the data cache. Erroneous values are inserted randomly in the cache accesses and propagated during the execution of the applications. At the same time, the proposed error metrics for each of the applications are scanned. Every mismatch of the values between the correct simulation and the erroneous simulation is counted as an application error for the corresponding application metric. We observe some

error metrics reacting more sensitively to hardware faults than others. As discussed in Section 3, the error metrics could be classified as *volatile* and *non-volatile* errors. For example, any error in the RouteTable entries is permanent and affects the system severely. On the other hand, an error in the *ttl* value of a packet during routing is limited to the corresponding packet, hence is a volatile error. However, we do not make such a distinction in our simulations: regardless of the source of an application error, we simply observe the output and capture the change in the application output. These changes may be caused by volatile errors or nonvolatile errors. From our perspective, we are only interested in measuring the change in the application output for a given hardware fault rate and this is independent of the sources of the errors.

3.3.3 Fallibility Factor

We need to introduce a measurement index to analyze the effect of hardware faults on the networking applications. Since the processor is going to make errors, traditional approaches such as delay, energy, or energy-delay product would be insufficient. We define the metric *fallibility* as the probability of the processor making an error for the application. One can use the number of hardware faults that are not detected to measure the fallibility. However, due to the application-specific nature of our target architectures, we use application errors in the fallibility factor as discussed in Section 4. Particularly, fallibility corresponds to the fraction of packets that have any type of errors. Note that even if the packet is correctly forwarded, it can still contribute to the fallibility rate. For

example, if the *ttl* value of the packet is different than what it would be in case of correct execution, we consider the packet to have an error.

3.3.4 Fatal Error Probability

We pay special attention to fatal errors. Since fatal errors prevent other packets from being processed[4], we calculate the number of packets successfully processed until the occurrence of a fatal error. The reported fallibility factors are based on this number. We also report the probability of a fatal error in addition to the fallibility factor. Particularly, we record the probability of a fatal error with increasing error introduction rate. Increased hardware faults make the system more susceptible to termination. As a result fewer packets can be processed successfully at higher error introduction rate.

The cause of the fatal errors can be attributed to several factors and we can classify the errors accordingly. A fatal error may occur due to an unimplemented system call or an access to restricted or non-existent memory location. The system crashes if one of these fatal errors occurs. However, the system may run into an infinite loop because of an error. We classify this fatal error as the "silent error".

We propose different remedies for the fatal errors depending on their nature. The destructive errors can be taken care of by higher levels of the system (e.g. operating system) that reset the system to a stable state to prevent a system

---

[4] Majority the fatal errors we have observed during our simulations are caused by the execution getting into an infinite loop.

crash. However, to prevent the "silent error", we can implement the check-pointing scheme to prevent the system from running into an infinite loop.

3.3.5 Comparison Metric

We need to introduce a measurement index to determine the "optimal" point of operation. Since, the processor is going to make errors, traditional approaches such as delay, energy, or energy-delay product would be insufficient. We define the metric *energy-delay-fallibility* product, which is the product of the energy consumption, the execution cycles of the application, and the "*fallibility*" factor of the processor. The energy consumption is the energy consumed in the whole processor during the execution of the application. Particularly, fallibility corresponds to the fraction of packets that have any type of errors. We also pay special attention to the fatal errors. Since fatal errors prevent other packets to be processed[5], we calculate the number of packets successfully processed till the occurrence of a fatal error. The reported energy-delay-fallibility factors are based on this number. We also report the probability of a fatal error in addition to the energy-delay-fallibility product. Particularly, we record the probability of a fatal error with increasing clock frequency. Increased clock frequency makes system more susceptible to termination. As a result less number of packets can be processed successfully at higher clock frequency.

---

[5] Majority the fatal errors we have observed during our simulations are because the execution gets stuck in an infinite loop. For such an error, the processor can be modified such that we can recover from the error.

Although we argue that the packet processors can have faults, frequent faults are certainly undesirable considering the system behavior. Therefore, instead of giving the same weight to each component in energy-delay-fallibility product, one can give more weight to the fallibility. Particularly, the product can be calculated as $energy^k$-$delay^m$-$fallibility^n$ according to the needs of the architecture. In our studies, since delay and fallibility are more important than energy, we set k to 1, m to 2, and n to 2. The energy-delay-fallibility product can be defined for a single component (e.g., cache). However, in this work, we measure the metric for the applications.

3.4. Experimental Results

3.4.1 Simulation Environment

We use the SimpleScalar/ARM [42] for our simulations. We modified the processor configuration to model a processor similar to execution cores in a variety of Network Processor architectures. Particularly, we simulate a processor similar to StrongARM 110 with 4 KB, direct-mapped L1 data and instruction caches with 32-byte line- size, and a 128 KB, 4-way set-associative unified L2 cache with a 128-byte line-size. The level 1 data cache has 2-cycle latency and the level 2 cache latency is 15 cycles. We first modified the applications to mark the values of data structures mentioned in the previous chapter. Then, we have modified the simulator to introduce random faults into the execution and to simulate the effects of the introduced faults. We chose an initial fault probability of $2.59 * 10^{-7}$ per bit (in accordance with the formula (2.4)). This fault rate is similar to the rates

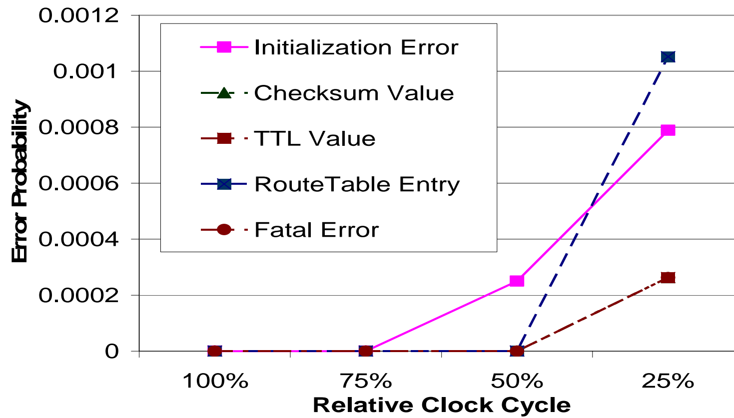
reported by Shivakumar et al. [55]. The probability of a two-bit fault is set to $2.59*10^{-9}$, and the probability of three-bit faults is $2.59*10^{-10}$ in accordance with reported correlation between single-bit and multiple bit faults [56]. For the higher clock rates, we increase the fault rate in steps according to formula (Section 2.4).

3.4.2 Application Error Behavior

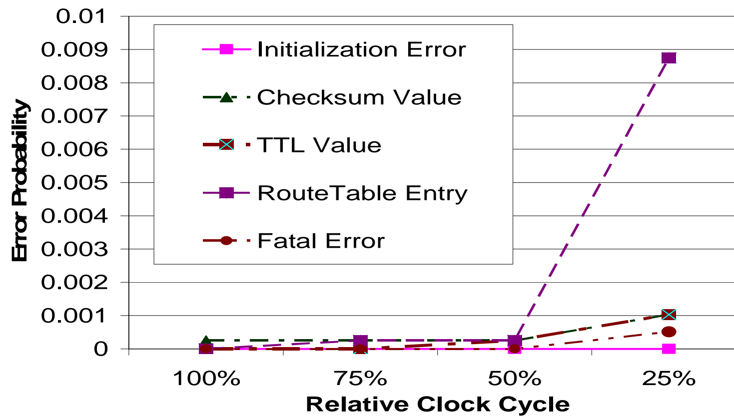
This section describes the simulation results observed for the networking applications. The experiments in this section measure the effect of different fault rates on the data structures discussed in Section 2.4.

Figure 3.2 presents the results for the ROUTE application. For the results presented in Figure 3.2(a), we only introduce faults during the control plane tasks. Similarly, for the results in Figure 3.2(b), faults are introduced only during data plane tasks. For the results in Figure 3.2(c), faults are introduced during both the control plane and data plane tasks. Intuitively, the faults in the control plane tasks should have significantly more effect on the application behavior. This can be observed for initialization error when Figure 3.2(a) and Figure 3.2(b) are compared. However, for most error types, the difference is not drastic. This behavior is due to the shorter length of the control plane tasks compared to that of the data plane tasks. Therefore, although each fault happening during the control plane tasks has larger impact on the error rate compared to the faults during data plane tasks, the overall impact of varying the clock rate during the control plane tasks is not drastically more on the application errors. This is an encouraging result, because in many cases the processor will not have information about the

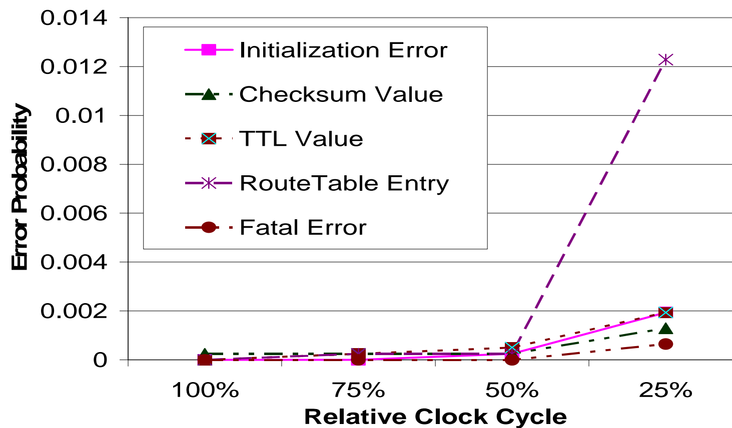
type of task it is executing. Hence, it might be complicated to have different clock rates for different tasks. Since the results indicate that the effect of faults during control plane tasks is tolerable, we can "safely" vary the clock frequency.



(a) Faults introduced in control plane

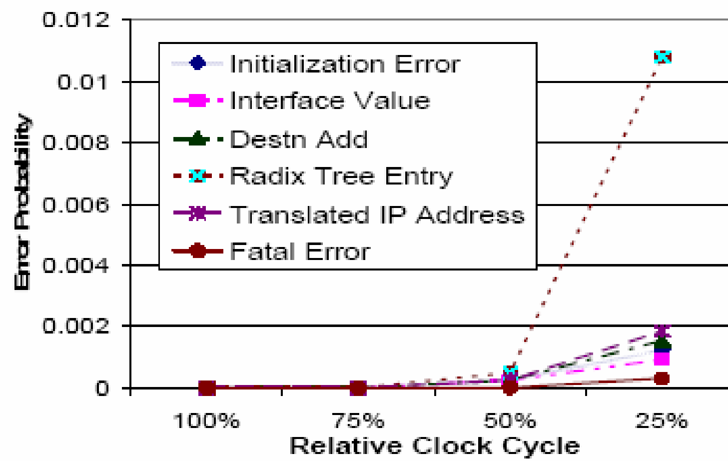


(b) Faults introduced in data plane

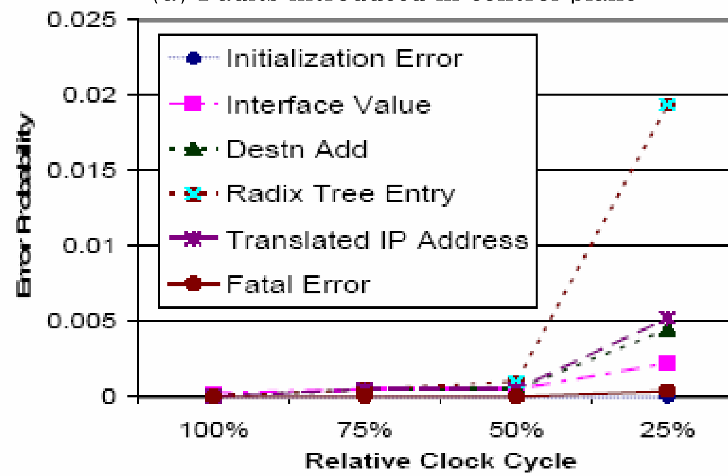


(c) Faults introduced in both data and control planes

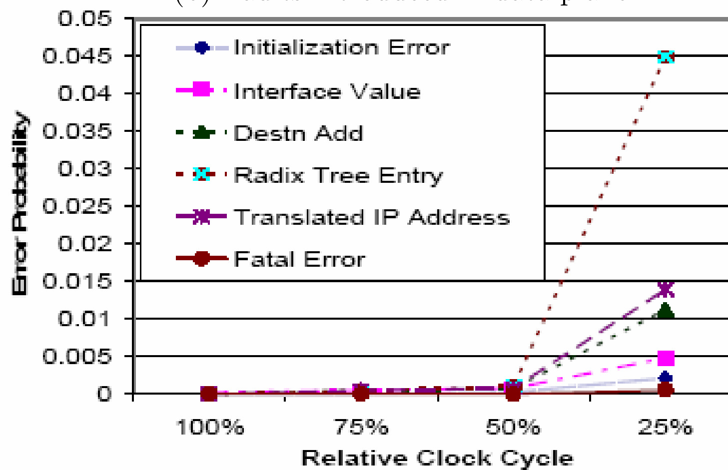
Figure 3.2. *Error Probability of ROUTE application*



(a) Faults introduced in control plane



(b) Faults introduced in data plane



(c) Faults introduced in both data and control planes

Figure 3.3. *Error Probability of NAT application*

Figure 3.3 presents the results for the NAT application. Similar trends can be observed for this application as well. Particularly for the NAT application we see that errors due to faults during data plane tasks have more impact on the application behavior than the faults during control plane tasks. The results for the remainder of the applications are not presented due to their similarities with the presented results. However, all of them show identical characteristics of the applications under erroneous execution. Overall, all the applications can sustain faults to varying extents. For smaller fault rates we observed the execution of the application without any observable error in the data structures and the application output. For larger fault rates, on the other hand, we encountered fatal errors and errors in the data structure values.

3.4.3 Fallibility Factor

In almost all the applications, we see that increasing the hardware fault rate increases the fallibility factor. For lower fault introduction rate, we observe a negligible change in fallibility factor, suggesting that the applications are not affected by the hardware faults. However, with the increase in the fault probabilities, the applications become vulnerable, which is properly reflected in the fallibility factor values. We recorded the probability of different application errors for each of the seven NetBench applications. The fallibility factor is obtained by summing up the probability of all application errors for each application. Table 3-A gives us a common framework to compare the behavior of the network applications subjected under hardware faults. The MD5 application shows

maximum sensitivity towards errors. It has a fallibility factor of 0.261 when the error introduction rate is highest (25% relative clock frequency). This rate means that on average, in 26.1% of the packets differ from the execution without any faults.

Table 3-A. *Fallibility Factor of Different Applications*

Appln.	Fallibility Factor			
	Static Overclocking Rate – Relative clock Freq (C_r)			
	100%	75%	50%	25%
CRC	0.0023	0.0038	0.0073	0.0524
Tl	0.0010	0.0063	0.0159	0.1350
ROUTE	0.0003	0.0008	0.0013	0.0175
DRR	0.0000	0.0010	0.0023	0.0076
NAT	0.0003	0.0020	0.0035	0.0770
MD5	0.0000	0.0115	0.0552	0.2610
URL	0.0003	0.0013	0.0025	0.0177

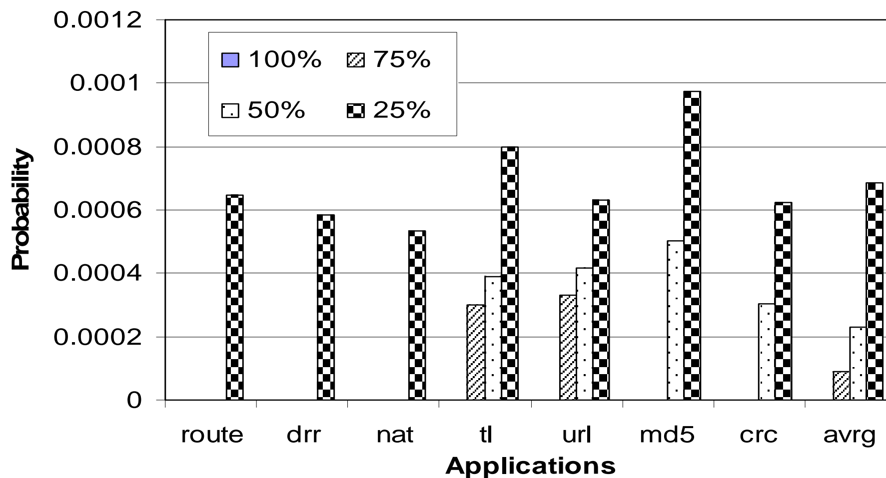


Figure 3.4. *Fatal error probabilities for different clock rates.*

3.4.4 Fatal Error Probability Measurements

We recorded the probability of a fatal error with increased clock frequency. Unlike other errors, fatal errors may destroy the system integrity. This prompts to

ensure that the clock frequency should not reach a value that may result a high probability of fatal error. Figure 3.4 depicts the fatal error probability for different applications when there is no error detection scheme employed. Similar to the fallibility results, we see that the fatal error probability is zero for smaller increases in the clock rate. As we exceed 100% increase in the clock rate, we start seeing an impact on the fatal error probability. Note that the fatal error probabilities in Figure 3.4 are measured for the base architecture, which does not employ any error detection scheme. Error detection schemes reduce the probability of fatal errors dramatically. In fact, during the simulations of the architectures with error detection, we have never encountered a fatal error.

3.4.5 Energy-Delay-Fallibility Measurements

The simulations presented in this section introduce faults during both the control plane and the data plane. As we have discussed in Section 3.3.5, different techniques are compared using the energy-delay^2-fallibility^2 product. To measure the energy consumed during the applications we use three models. For the energy consumption of the overall processor, we used the results presented by Montanaro et al. [57]. The energy consumed by the caches when they are operated with full frequency is found using CACTI [58]. When the clock frequency is increased, the voltage swing decreases. The energy consumed by the cache linearly shrinks with this decrease in the voltage swing. Therefore, we used the model presented in Figure 2.2 to find the relative voltage swing for different clock rates. Particularly, the energy consumed by the cache reduces by 45%, 19%, and 6% for relative clock

rates of 0.25, 0.5, and 0.75, respectively. To estimate the energy consumed by the error detection scheme, we use the results presented by Phelan [59]. The level-1 data cache consumes 16% of the overall chip energy. Parity increases the energy consumed during reads by 23%. Similarly, the energy consumed during writes increases by 36%. We assumed that each word (32-bits) is protected by a single parity bit. To measure the delay in the applications, we calculate the average number of cycles spend for each packet. Note that we cannot use the total number of execution cycles, because some simulations do not finish to completion due to fatal errors. The fallibility factor is calculated as explained in Section 2.5.1.

Results for the ROUTE application are summarized in Figure 3.5. For ROUTE application, we see that the best technique is the static technique with 50% relative clock cycle when two-strike recovery is used. For the CRC application (Figure 3.6), on the other hand, the best configuration is the dynamic frequency adaptation with three-strike recovery. When we compare these two applications, we see that CRC is more resilient to faults, because due to its streaming nature it already has a large cache miss rate. Therefore, additional cache accesses due to errors have less effect on the execution time. As explained in Section 3.3.1, three-strike eliminates some of the incorrect accesses to the level 2 cache that might happen by the two-strike scheme. Therefore, three-strike improves the performance for the CRC application because it reduces the pressure on the level 2 cache.

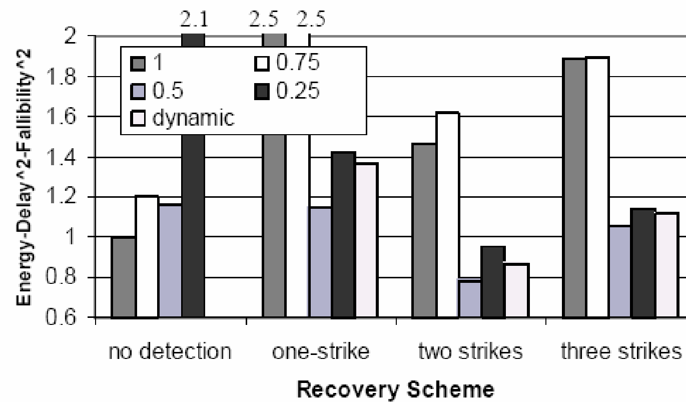


Figure 3.5. *Energy-delay2-fallibility2 product for the simulated configurations, the dynamic configuration, and the static configuration with Cr = 1, 0.75, 0.5, and 0.25 for the ROUTE application. The bars represent the relative energy-delay2-fallibility2 product with respect to Cr = 1 with no-detection.*

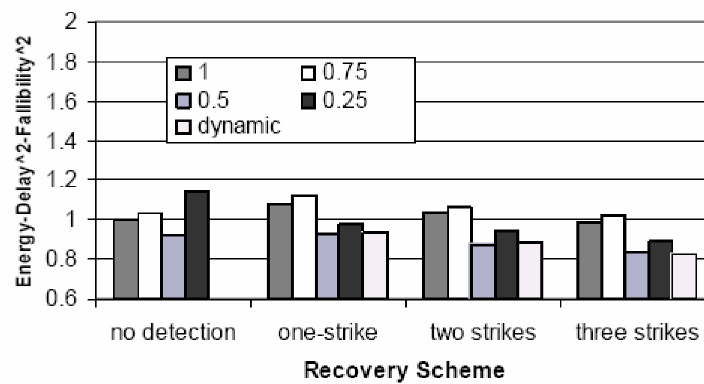


Figure 3.6. *Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the CRC application.*

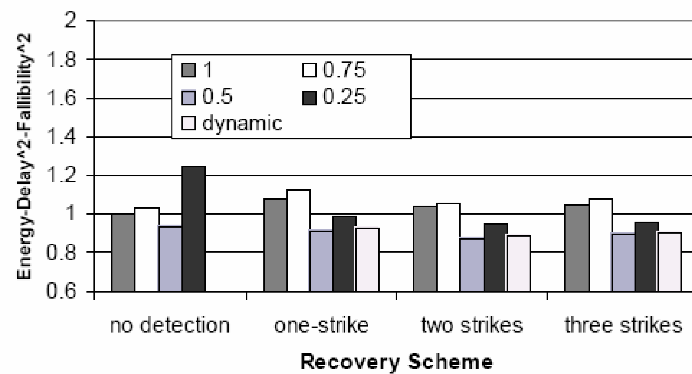


Figure 3.7. *Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the MD5 application.*

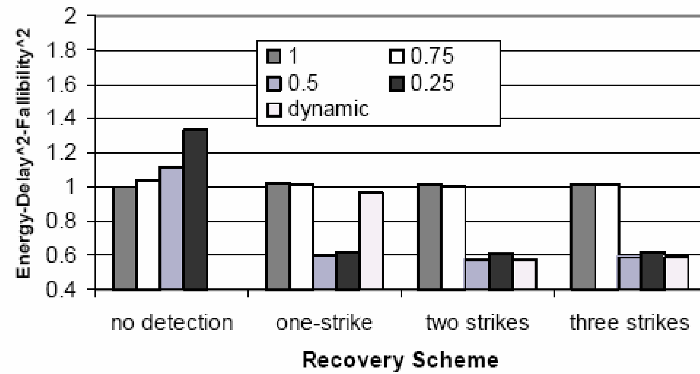


Figure 3.8. *Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the TL application.*

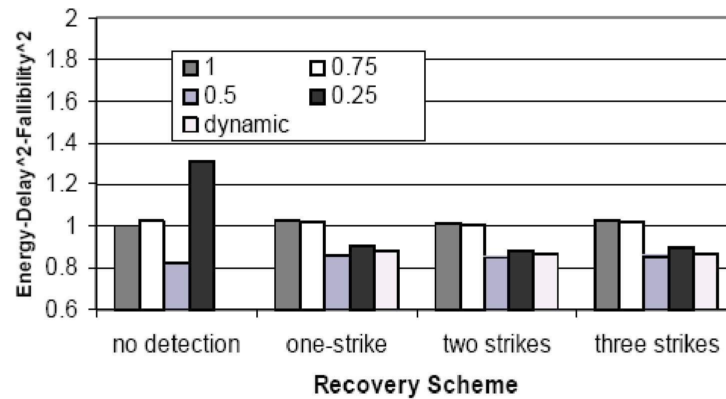


Figure 3.9. *Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the DRR application.*

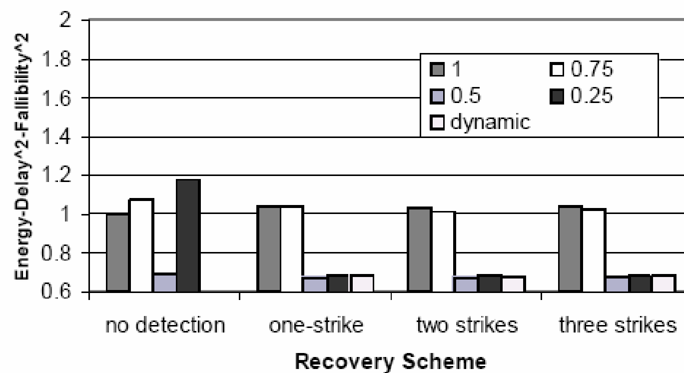


Figure 3.10. *Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the NAT application.*

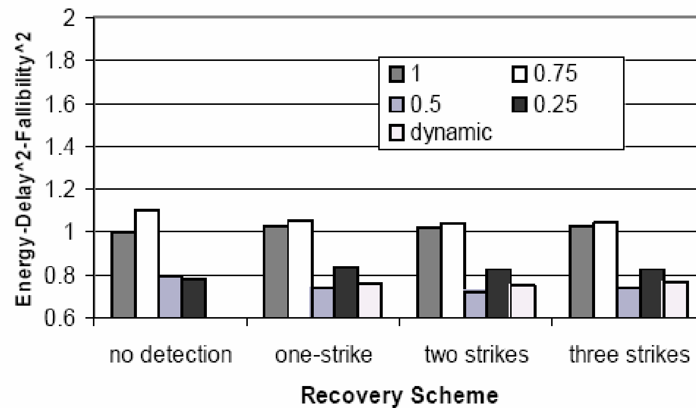


Figure 3.11. *Energy-delay²-fallibility² product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 for the URL application.*

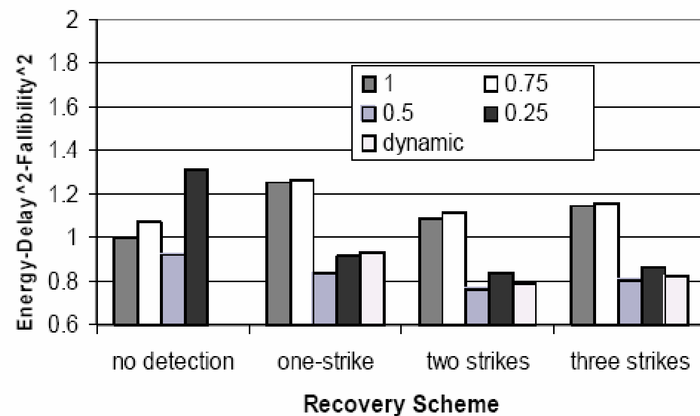


Figure 3.12. *Energy-delay²-fallibility² product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25 the average application.*

Figure 3.7 and Figure 3.8 present the results for the MD5 and TL applications, respectively. We see that similar to the ROUTE application, the static technique with 50% relative clock cycle and two-strike recovery scheme gives the best result. For the TL application, we see that the energy-delay^2-fallibility^2 product is reduced by as much as 43%. TL application has a large fraction of load

instructions. Therefore, reducing the cache access latency has a significant impact on the overall performance.

One interesting result with the TL application (Figure 3.8) is the inability of the dynamic scheme to reduce the energy-delay^2-fallibility^2 product for the one-strike scheme. The reason for this is due to some initial errors, the dynamic scheme gets late into the 0.5 region. Since the total number of instructions executed for this application is small, the overall energy-delay^2-fallibility^2 remains high. The results presented in Figure 3.9 and Figure 3.10 are for the DRR and the NAT applications.

Figure 3.11 presents the results for the URL application. Figure 3.12, on the other hand, gives the average of all the simulated applications. Overall, we see that the static technique with 50% relative clock cycle and two-strike recovery scheme gives the best result reducing the energy-delay^2-fallibility^2 product by 24%. This is partially an artifact of the steps we have selected for the clock frequency. Although when we set C_r to 0.25, we see a significant reduction in the energy consumption, we also see a sharp increase in the error rates. Therefore, $C_r = 0.5$ almost always performs better than the $C_r = 0.25$. As a result, the dynamic scheme also stays mostly in the $C_r = 0.5$ region and hence does not perform better than the static scheme. Note that if we do not consider the errors, the static approach with $C_r = 0.5$ and two-strike recovery scheme reduces the energy-delay product of the processor by 17%, and the energy-delay^2 product by 26%.

In almost all the applications, we see that without the error detection, increasing the clock frequency increases the energy-delay^2-fallibility^2. The reasons for this are two-fold. First, we take the square of the fallibility in our metric. Since we increase the fallibility factor when we increase the clock frequency, there is a significant increase in our metric. Second, we see that errors usually increase the number of execution cycles. There are two reasons for this. First, erroneous load operations usually result in misses in the cache. More importantly, we see that the number of instructions executed also increases with the errors. This is mostly due to the loops. If one of the values that affect the completion criteria changes, we see that in most cases the number of iterations increase.

3.5. Previous Work on Resilient Architectures

One class of related work is in the area of fault tolerance. Traditionally, fault tolerance has caught attention in the context of environments with heavy concentration of alpha-particles and atmospheric neutrons [60]. Transient faults induced by these particles are shown to decrease the reliability of processors [61]. Another area where there has been a strong emphasis on reliability is circuit verification, which is an important problem in IC fabrication. Techniques exist to study potential errors in the pre-silicon [62] stage and also subsequent to the fabrication process [63]. More recently, designing computer systems for resiliency [64] to transient faults has gained greater significance due to the combined effect of higher integration densities, lower voltages, and faster clock frequencies. There

have been various studies utilizing redundancy to increase robustness for SMT processors [65, 66], for superscalar processors [67], and for CMPs [68]. All of these techniques aim to increase robustness. Our approach, on the other hand, reduces it. Although this might seem controversial at a glance, our motivations are similar to these studies: correctness cannot be achieved by optimizations only at the circuit level. However, we propose to deal with the errors at the higher levels instead of trying to eliminate them.

Validation methods such as fault injection are particularly attractive for estimating the dependability of computer systems [69]. Mukherjee et al. introduces the architectural vulnerability factor (AVF) for various processor components [70]. However, we are not aware of any study that investigates the application-level behavior of networking programs under hardware faults. More importantly, these studies still do not allow an incorrect execution of the program as we propose in this work. Austin introduces DIVA, which is a method for enforcing correctness in processors which can make mistakes because of the lack of complete verification [6]. DIVA still aims to achieve correctness, whereas in this approach we reduce the probability of correct execution.

B. Statistical Task Allocation in Multicore Network Processors

Chip multiprocessor designs are the most common types of architectures seen in Network Processors. In such designs, there are numerous approaches to implement interconnects. However, due to the trends in scalability of global interconnects, we observe a shift towards '*systolic array*' architectures where different cores are connected through point-to-point links and the packets are processed in a pipelined fashion. Intel's IXP and Cisco's Toaster processor fall into this category. As the Network Processors are used to implement increasingly complicated architectures, task distribution among the cores is becoming an important problem. In this chapter, we propose a new task allocation analysis scheme. This scheme relies on the inherent modular nature of the networking applications and intelligently distributes modules among different execution cores. An important problem that needs to be tackled in this aspect is the variation of execution times of the modules. To address this problem, we have developed a technique that uses the probability distribution of the execution times of different modules in the networking applications. Furthermore, we have used overclocking technique to optimize the performance of the allocated tasks. The combination of statistical distribution of modules and the overclocking result in significant performance improvements for representative architectures.

One of the most important bottlenecks for CMP processors in general and particularly the Network Processor architectures, is the low scalability of the

interconnect networks. Although increasing the number of cores in the processor is desirable to take advantage of the parallelism in the application, developing an interconnect network to achieve efficient communication among cores becomes complicated as the number of nodes is increased. Therefore, with the new generation network processor architectures, we are seeing an increased emphasis on local communication. For example, the Intel IXP 28xx [71, 72] architectures utilize neighbor-to-neighbor links in addition to the global communication structures. In such architectures, the utilization of the local communication links is arguably the most important factor in determining the performance of the application. With the increasing link speeds and the changes in the target applications, it is expected that the number of execution cores in the processors will increase. Therefore, high utilization of local communication links will become an obligation to achieve the desired scalability in next-generation network processors. Clearly, the key factor that determines the communication behavior is the task distribution. In other words, the mapping of application functionality onto multiprocessing elements must be performed intelligently to achieve the desired level of performance. In most of the existing architectures, this task is left to the user. With the increasing complexity of the architectures, this expectation from the user becomes limiting and an automated task distribution scheme is highly desirable. In this chapter, we propose a solution to this problem. Particularly, we present an automated task allocation analysis scheme for network processors. We utilize the modular nature observed in majority of networking applications. First, we divide the applications

into modules similar to CLICK [73] and NP-Click [74] environments. Then, we profile the applications using a representative workload and perform a statistical analysis on the behavior of different modules in the application. This analysis provides us with the distribution of the execution times of different modules. Then, we utilize this probability distribution information to allocate tasks among different execution cores of the network processors. In addition to utilizing this information to decide on task distribution, we also make decisions about which modules should be replicated based on the analysis. Specifically, our contributions in this chapter are as follows:

- We analyze the probability distribution of packet processing elements in a modular networking applications,
- We present an intelligent methodology to allocate tasks among different processor cores of a chip multiprocessor,
- We show experimental results investigating the impact of smart task allocation.

Our main goal is to reduce the effect of the variation in the execution times of the packets. To be more precise, we would like to schedule the tasks such that the effects of variation will be minimized. The variation in the execution times is an inherent property of computers. This is particularly true for CMPs, where different cores are competing for a set of global resources (e.g., shared bus or the shared memory). In addition, there is data-dependent variation, i.e., depending on

the input the execution time may vary. For example, a loop might be executed for different number of iterations based on the input data. This uncertainty is even more pressing if the cores implement multithreading (as commonly done for most Network Processor architectures). The order of the thread selection and the order of the packet arrival are likely to have a significant impact on the time of completion for a single thread. This inherent variation in execution time is an important reason for the complexity of task distribution. Consider a distribution where $Task_A$ is executed on $Processor_A$ and the results of this task is fed into $Task_B$, which is executing on $Processor_B$. If the execution times of these tasks were constant, we could possibly arrange them such that the tasks allocated to $Processor_A$ and $Processor_B$ will be equal execution time and hence both processors will have 100% utilization. However, consider in this case, that $Task_A$ is prolonged. In that case, $Processor_B$ will complete the execution of $Task_B$ and will remain idle until $Task_A$ is finished by $Processor_A$. This will certainly reduce the overall utilization of the resources and hence will result in degradation of performance. Therefore, a task distribution scheme should consider the variation in execution times while making resource allocation decisions. In this chapter, we describe such a scheme.

We discuss the modular property of networking applications in Section 3.6. Section 3.7 and 3.8 analyzes the implementation problems in networking processor domain and explores possible solution. Section 3.9 gives an overview of the applications used in our studies. Section 3.10 presents the statistical analysis of the

packet processing of modules in different networking applications. In Section 3.11, we discuss our novel task allocation scheme and different optimizations implemented on it. Section 3.12 presents the experimental results. In Section 3.13, we discuss the related work followed by concluding remarks in Section 3.14.

3.6. Modularity in Network Applications

The Network Processor (NP) designers utilize two important properties of networking applications. First, these applications consume and produce well-defined data segments (network packets). This property leads the designers to utilize intelligent memory controllers specifically designed to move packet data to/from and within the processor. Secondly, for many of the networking applications, though not all, these packets can be processed independently. Therefore, there is a large amount of data level parallelism available in the applications. The designers take advantage of this fact with the use of multithreading and with multiple execution cores. Almost all of the NPs use a variation of multithreading and have several execution cores.

Another important aspect that needs to be highlighted is the trend in the NP architectures. Each new NP generation employs more execution cores than their predecessors. Therefore, traditional communication structures between these execution cores (global buses or cross-bar based fabrics) become less effective. Many of the newer NPs employ special neighbor-to-neighbor communication (systolic array) or enhanced interconnection networks to reduce the need for

accessing global structures. In such systems, effective task allocation becomes particularly hard even for the most experienced programmers.

Table 3-B. *Important characteristics of representative Network Processor Designs: exec. cores is the number of execution cores, and parallelism technique is the technique(s) used for task or instruction level parallelism (MT: Multi-Threading, VLIW: Very-Long Instruction Word) in the execution cores*

Processor	# of cores	Parallelism technique
Agere PayloadPlus	3	MT, VLIW
AMMC (MMC) nP7250	2	MT
Bay Microsystems Chespeake	2	MT
Broadcom BCM-1250	2	Superscalar
Cavium Octeon	16	MT
Cisco Toaster	16	VLIW
EZChip	~40	MT
Freescale C-5	16	MT
Hifn 5NP4G	16	MT
Intel IXP2800	16	MT
Intel IXP1200	6	MT
PMC-Sierra RM9000	2	Superscalar
Vitesse (Sitera) IQ1200	4	MT
Wintegra WinPath2	6	MT
Xelerated Xelerator X11	360	VLIW, MT

In the past, modular routers have gained much focus due to their ease of designing. CLICK [73] and Baker [75] are examples of domain specific languages designed for describing networking applications. We design our framework based on the CLICK framework. CLICK is a flexible, software modular architecture, which can build routers from fine-grained components. Each of these components, known as *element*, performs a simple task, such as decrementing an IP packet's time-to-live (TTL) or IP header checking. They can easily be extended to do complicated tasks (IP lookup, NAT). To build a router configuration, the user chooses a collection of elements and connects them into a directed flow graph. The nodes being elements, the connections between those elements represent a

forwarding path. Click configuration scripts are written in a simple language with two important constructs: declarations create elements, and connections say how they should be connected.

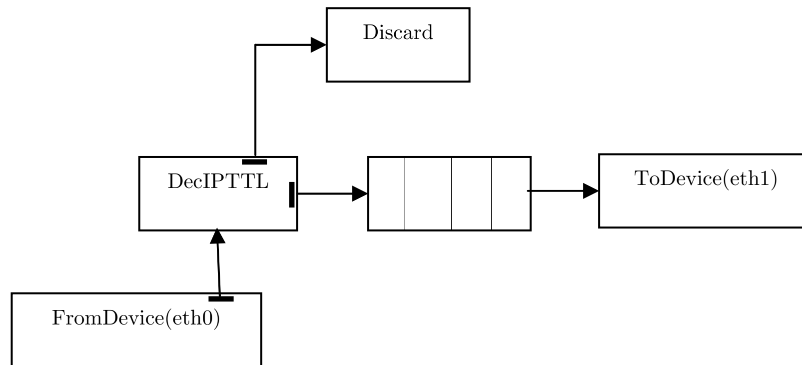


Figure 3.13. *Click configuration for TTL decrement*

The Click language is wholly declarative. It specifies what elements to create and how they should be connected, not how to process packets procedurally. Router manipulation tools can take advantage of these properties to optimize router configurations offline or prove simple properties about them. The main goals behind the Click language are usability and extensibility. Figure 3.13 shows a Click diagram of a simple configuration that checks the TTL value of a packet. It forwards the packet if the TTL value is non-negative. Otherwise, the packet is discarded. More details about the programming model is available in the next section.

3.7. Implementation Gap

Click is a natural environment for describing packet processing applications. It is expected that networking applications should be mapped directly into a

network processor. However, there is currently a large gap between Click and the low level programming interface the network processors expose.

Click proposes a simple yet powerful concept of push-and-pull communication between elements that communicate only via passing packets, coupled with the rich library of elements. This is a natural abstraction that aids designers in creating a functional description of their application. Note that, this is in stark contrast to the main concepts required to program a network processor. While implementing an application on this device, it is the programmer's responsibility to effectively partition an application across the different execution cores, make use of special-purpose hardware, effectively arbitrate shared resources, and communicate with peripherals. This mismatch of concerns between the application model and target architecture is known as the *implementation gap* (see Figure 8). To facilitate bridging this gap, there is a need for an intermediate layer, called a ***programming model.*** It presents a powerful abstraction of the underlying architecture while providing a natural way of describing applications.

3.8. Implementation Gap Closure Approaches

Different approaches have been proposed to solve the *implementation gap.* Works in this area can be categorized into four major areas: library of application components, programming language-based, refinement from formal models of computation (MOCs), and run-time systems. In this section, we describe and evaluate these alternatives.

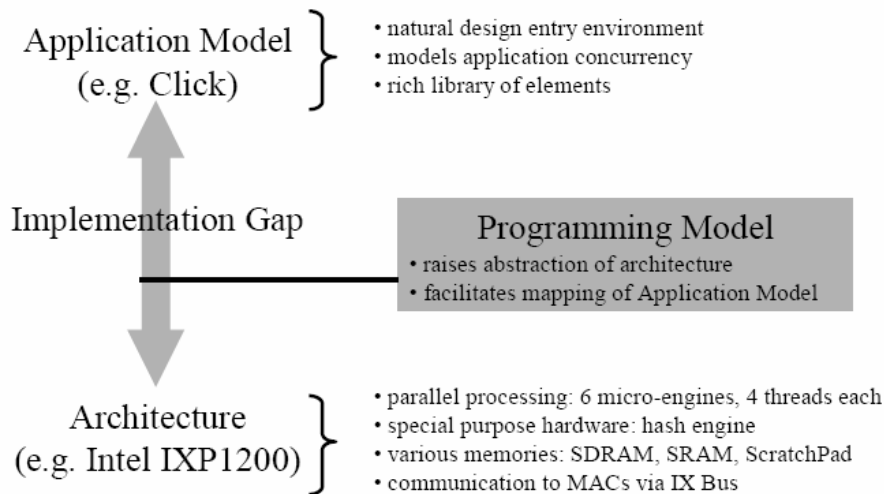


Figure 3.14. *Implementation Gap*

The *library of application components approach* exports a collection of manually designed blocks to the application programmer, who stitches these together to create his application. The advantage of such an approach is a better mapping to the underlying hardware, since the components are hand-coded. In addition, these components implement an abstraction that is natural for an application writer as the components are often similar to application model primitives. The disadvantage of this approach is the need to implement every element of the library by hand.

If only a limited number of library elements are needed, this approach may be successful. However, in practice, we suspect a large number of elements are needed as application diversity grows. This problem is further compounded when a number of variants of each library element are needed.

A *programming language approach* utilizes a programming language that can be compiled to the target architecture. With this approach, a compiler needs to be written only once for the target architecture and all compiler optimizations can be applied to all applications that are written for the architecture. The principal difficulty with this approach is the requirement to compile to heterogeneous architectures with multiple processors, special-purpose hardware, numerous task-specific memories, and various buses. In addition, the programming abstraction required to effectively create a compiler for such architectures would likely force the programming language to include many architectural concepts that would be unnatural for the application programmer. Examples of this alternative include the numerous projects that have altered the C programming language by exposing architectural features [76, 77].

Another class of approaches uses *refinement from formal models of computation* (MOCs) to implement applications. Models of computation define formal semantics for communication and concurrency. Examples of common MOCs include Kahn Process Networks [78] and discrete-event. Because they require applications to be described in an MOC, these approaches are able to prove properties of the application (such as maximum queue sizes required and static schedules that satisfy timing constraints). This class of solutions also emphasizes application modeling and simulation [79]. The disadvantage of this method is that implementation on heterogeneous architectures is inefficient because most implementation paths require significant compiler support. As an example,

Edwards [80] has written a compiler to implement designs described in Esterel, a language that implements the synchronous/reactive MOC [13]. However, his work generates C code and relies on a C compiler for implementation on target architectures. In addition, the MOCs used by these approaches may not be natural design entry environments. For example POLIS requires all applications to be expressed in co-design finite state machines [79].

Run-time systems offer another category of solutions to the *implementation gap*. Run-time systems introduce dynamic operation (e.g. thread scheduling) that enables additional freedom in implementation. Dynamic operation can also be used to present the programmer with an abstraction of the underlying architecture (e.g. a view of infinite resources). While run-time systems are necessary for general-purpose computation, for many data-oriented embedded applications (like data plane processing) they introduce additional overhead at run-time. Additionally, some ASIP architectures have included hardware constructs to subsume simple run-time system tasks like thread scheduling on the IXP1200 and inter-processor communication (ring buffers on the Intel IXP2800 [72]). Examples of this approach include VxWorks [81] and the programming interface for the Broadcom Calisto [82].

Based on the trade-offs between the above approaches, Shah et al. proposed a new programming model named as NP-Click [74]. It is an extended programming model based on Click Router and implemented on the Intel IXP1200. It is a combination of an efficient abstraction of the network processor with features of a

domain-specific language for networking. The result is a natural abstraction that enables programmers to quickly write efficient code. It facilitates the difficulties of programming network processors by taking advantage of hardware parallelism, arbitration of shared resources, and efficient data layout.

We have used this programming model to perform a statistical task allocation in CMP style network processors. As discussed later, the inherent modularity of Click configurations facilitate my objective of optimized task allocation. We have used statistical data in conjunction with the modularity information to perform an effective task allocation among different execution cores of a network processor.

3.9. Applications

We explore the effectiveness of our task allocation techniques by using it to schedule four representative networking applications. This section describes the application we simulate.

IPV4Router: We implement the data plane of an 8 port Fast Ethernet IP Version 4 router [83]. This application is based on the network processor benchmark specified by Tsai et al. [84]. A packet arriving on port P is to be examined and forwarded on a different port P'. We use a static lookup table to decide the next-hop location. It is determined through a longest prefix match (LPM) on the IPv4 destination address field. The packet header and payload are

checked for validity and packet header fields' checksum and TTL are updated. Figure 35 shows this particular Click configuration tree.

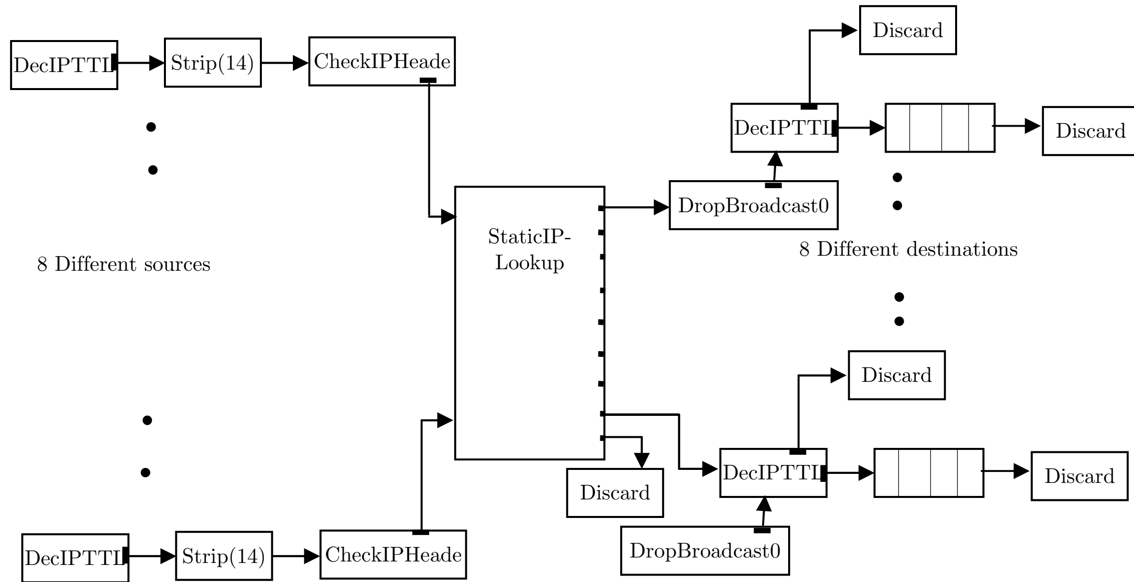


Figure 3.15. *Click configuration tree for the IPV4Router application*

DRR: We extend the IP router demonstrated in the Click Modular Router project. The router that forwards unicast packets in nearly full compliance with the standards [83, 85, 86]. We introduce a queue which introduces packet by pulling from a set of infinite packet source using deficit round robin (*DRR*) scheduling [87].

RED: Random early detection is more likely to drop packets when there is network congestion; when there are many packets in the queue servicing that link. The RED element therefore queries router queue lengths when deciding whether to drop a packet. For this application, we extend the Click IP router to handle specialized routing tasks. Particularly, a complex IP router performs the following tasks: two parallel T1 links to a backbone, between which traffic should be load

balanced; division of traffic into two priority levels; fairness among the connections within each priority level; RED dropping driven by the total number of packets queued. Click's modular scheduling, queuing and dropping policy elements are used in this application.

HOME_NODE: This application imitates an active home node in a network. The home node proxy-ARPs for the mobile node, decapsulates packets from the remote node, sending them onto the local network, and perform IP encapsulation for packets destined for the mobile node. It also ensures that packets generated by the address 1.0.0.10 are properly encapsulated.

3.10. Probability Distribution of Packets

In this section, we discuss the probability distribution of the packet processing time in a Click modular application. For the sake of conciseness, we describe the results for only the *IPV4Router* application in detail. The rest of the applications are analyzed in the same fashion and their results are summarized at the end of this section in Table 3-D. The simulation environment used to gather the statistics is described in Section 3.12.1.

The *IPV4Router* application consists of 33 Click elements. It has five different basic elements – *Strip(8), CheckIPHeader(8),StaticIPLookup(1), DropBroadcasts(8), DecIPTTL(8)* [88]. Table 3-C shows a graphical representation of the Click description of the router and the relation between the basic elements (i.e., modules).

Table 3-C. *Probability Distribution of IPV4Router Elements*

	Mean (μ)	SD (σ)	Processing time threshold				
			μ	$\mu+\sigma$	$\mu+2.\sigma$	$\mu+3.\sigma$	$\mu+4.\sigma$
strip0	241.28	29.31	50	0.64	0.64	0.64	0
strip1	232	22.11	34.19	33.33	0	0	0
strip2	220.18	25.24	31.81	25	0	0	0
strip3	216.05	19.87	35.06	20	3.63	0	0
chkip0	713.01	59.77	50	0.64	0.64	0.64	0.64
chkip1	695.22	31.48	34.63	33.33	0.86	0.43	0
chkip2	695.49	25.09	27.59	25	0.97	0	0
chkip3	694.63	21.77	20	20	2.33	0	0
RtLkUp	336.56	266.88	20.03	20.03	10.01	0.03	0.03
DBC0	212.30	21.18	34.32	28.57	1.29	0.18	0.18
DBC1	197.42	18.51	51.29	26.94	25	0.64	0
DBC2	210.45	26.50	18.39	2.16	0	0	0
DBC3	205.47	17.40	32.83	14.28	14.28	0	0
DcTTL0	317.78	20.34	26.45	12.98	2.09	0	0
DcTTL1	320.33	21.10	35.71	26.62	0	0	0
DcTTL2	315.96	19.6	20.99	17.09	1.29	0	0
DcTTL3	314.77	18.26	19.85	14.28	0.55	0	0

We execute the configuration for 5000 packets. During this execution, we record the amount of time spent by each packet in different elements of the Click router. Using these statistics, we find the mean and the standard deviation of the execution times. Table 3-C presents the results for the 17 elements that have the longest execution times in the configuration. Once we extract the mean (μ) and standard deviation (σ) of processing time by each of the element, we compare them against the data recorded for each packet traversed through it. The columns in the Table 3-C record the percentage of packets that couldn't be processed within the slack given by the expression ($\mu+k.\sigma$), where k is a positive constant. This statistics is important for us, because it can be used as estimation for how the variation will effect the utilization. In other words, if we pipeline the tasks according to the mean only, a packet that takes longer than μ cycles to execute

will clog the pipeline and cause the utilization to decrease in the proceeding processor. The results indicate that variation can indeed become an important bottleneck. Particularly, if we only consider the average execution time in task distribution, 32% of the packets on average will not finished within the expected time and will likely cause performance degradation. In Section 3.12.2, we analyze the applications and present experimental results showing that for an 8 processor Network Processor, this variation can cause up to 23% underutilization of the processors.

Table 3-D. *Probability Distribution of Application Elements*

	Mean(μ)	SD(σ)	Processing time threshold				
			μ	μ+σ	μ+2.σ	μ+3.σ	μ+4.σ
DRR							
Cl1	351.20	24.13	55.35	9.81	0.88	0.13	0.13
DRRelem	17032.70	76778.25	45.23	0.13	0.13	0.13	0.13
IPCheck	31.66	38.14	13.82	0.25	0.25	0.25	0.00
RtLkUp	349.70	220.44	22.86	22.86	0.63	0.13	0.13
ChkPnt	19552.00	46879.27	7.64	7.64	7.64	7.64	0.33
FixIP	219.20	14.64	34.83	25.87	0.00	0.00	0.00
Frag	183.94	18.11	53.23	23.38	4.48	0.00	0.00
RED							
RED	834.80	142.74	39.33	6.78	6.61	5.72	0.06
StripHdr	204.00	9.89	25.08	9.17	7.17	5.08	2.33
GetIP	379.50	15.89	20.58	9.17	7.42	2.00	0.08
Strip2	209.00	13.60	23.83	16.00	9.67	0.17	0.00
IPEncap	469.70	17.65	33.50	15.50	3.00	0.17	0.17
SetIP	208.00	14.34	35.17	19.67	2.17	0.17	0.17
PrioSche	286.50	8.27	33.67	8.06	6.11	6.11	0.28
HOME_NODE							
Classifier	319.98	30.02	59.50	8.58	1.19	0.11	0.08
Strip1	213.90	9.96	26.25	7.83	2.75	1.08	0.33
CheckIP	695.80	20.31	41.67	12.50	0.08	0.08	0.08
StripHdr	225.70	10.15	32.58	11.67	1.58	0.08	0.08
GetIP	386.50	40.60	59.92	17.42	0.83	0.08	0.08

We analyze all the applications following the identical procedure. Table 3-D summarizes the statistical data obtained from different elements. Due to their similar nature, we report the data for a few representative elements in each application. Note that, the mean (μ) and standard deviation (σ) for different instances of the same element vary depending on the packet contents. Similar to the IPV4Router application, we see a large variation in the execution times of the modules for all the applications.

3.10.2 Aggregate Probability Distribution

Although we have seen a variation in the execution time of individual elements, the variations of different modules may cancel each other once they are formed into "stages" that will be executed in different processors. For example, if $element_1$ and $element_2$ are scheduled in a processor, if the execution time of $element_1$ is prolonged while the execution time of $element_2$ shortened, the overall variation in the execution time of the processor may remain constant. Therefore, we also analyzed the variation in the aggregate task execution. We divide the complete configuration tree into different stages. The boundary decision for each stage is made based on the data obtained from the probability distribution of individual elements. We use the expected execution time of the modules and form n stages that are of approximately equal size. Subsequently, we perform a probability analysis of packet processing in each of these stages. Table 3-E presents the probability distribution of the *IPV4Router* application when the processing path is divided into 4 stages. Note that, the selection of the number of

the stages is arbitrary, but we must highlight that the results are similar for different number of stages. The results indicate that the standard deviation for the aggregate elements is similar to the ones of the individual elements. Particularly, on average 29% of the packets will cause an execution time exceeding the mean.

Table 3-E. *Probability Distribution of IPV4Router Stages*

Stages	Mean (μ)	SD (σ)	Processing time threshold				
			μ	μ+σ	μ+2.σ	μ+3.σ	μ+4.σ
Stage0	227.38	24.14	35.06	20.00	3.64	0.00	0.00
Stage1	691.18	30.48	23.19	14.29	1.86	0.08	0.00
Stage2	500.43	29.52	27.18	24.31	5.66	0.11	0.11
Stage3	314.72	20.33	27.78	23.14	7.14	0.28	0.00

3.11. Statistical Task Allocation in NPs

In this section we describe how the statistical analysis is utilized during the assignment of tasks to execution cores (i.e., task allocation). The main objective of allocating tasks is to maximize the utilization of different execution cores of the network processor. This, in return, results in an increase in the throughput supported by the processor. In the following, we first describe our target architecture. Then, we present two module distribution schemes. The first assigns the tasks to the processors by simply considering the average execution time. The second one utilizes the standard deviation in addition to the average. If the number of execution cores exceeds the number of modules in an application, the modules need to be replicated. Section 6.3 describes how this replication can be performed effectively by taking advantage of the statistical information.

3.11.1 Architecture Description

In this work, we consider a systolic array architecture. In this architecture, the execution cores are arranged in a pipelined fashion. In other words, processors are logically aligned in a single dimension and each processor is connected to its left and right neighbors. In addition, for the communication patterns, which cannot be satisfied by the local links, a shared bus that connects all execution cores is utilized. Although generic, this architecture represents most of the existing Network Processor architectures. Our goal is to develop an automated method to distribute the tasks in an application uniformly over the cores. Once an execution core performs the task allocated to it, it forwards the processed packet as well as the necessary data to the next core.

3.11.2 Module Distribution

In this section, we describe how tasks or modules are distributed among execution cores. Note that, each Click element represents a conceptually simple computation. A *module* is defined as a subset of Click elements used in an application.

The Click configuration tree describes the flow of the application. When we combine the statistical data of individual Click elements along with the Click configuration tree, we have a tree structure depicting the estimated delay of a single packet processing. The overall flow of the processing task can be divided into stages. The objective while dividing the application into stages is to form a group of stages with equal expected delay. Note that, in our work a stage and a

module is synonymous. From this point onward, we would call each stage a module.

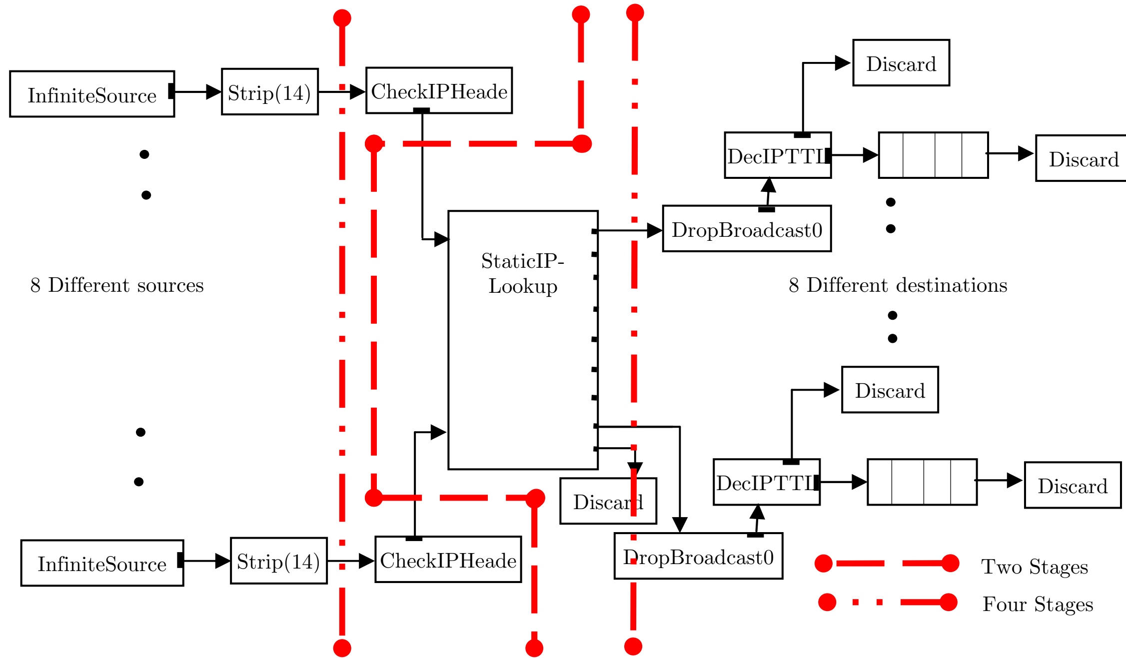


Figure 3.16. *Illustration of module distribution in IPV4Router application*

We utilize the average processing time of each element to form the modules. Assuming the tasks performed by each module is independent of each other; the average delay of each module is expected to be the sum of average processing time the Click elements. For a typical networking application, any particular packet would traverse one of the many alternate routes from start to end. We divide each of those paths into equal number of segments. The elements used in a particular stage of all the alternative paths form a single module. Note that, for a particular packet, only a subset of the all the elements in a module would be used.

To perform the module distribution, we use a three-step algorithm. In the first step, the total execution time of a packet is found. In the second step, the

stages are formed. In the third step, we perform local optimization to improve the task distribution. The algorithm starts from the root(s) of the Click configuration tree and traverses towards the leaf(s) of it. Each element is first assigned a weight equal to its expected mean execution time. Then, using a depth-first-search scheme, the algorithm assigns total weights to each element. Total weight of a child is equal to the sum of the total weight and the mean execution time of its parent. The selection of the child that will be traversed next is performed using the execution statistics. Particularly, the processing of a parent is followed by the processing of the most frequently executed child. This way, we follow the path that a packet would follow if there were no exceptions. If an element is visited before, its total weight will not be changed. The maximum total weight among the leaf nodes is selected as the execution time of a packet. Then, the maximum execution time is divided into the number of stages required (for a 4-core processors, the number of stages is equal to 4). This gives us the expected execution time for each stage. In the second stage of the algorithm, we again start from the root(s). Particularly, all the root(s) are first placed on stage 0. Then, the children of the root(s) are one by one added to the current stage until the total expected execution time of the stage reaches the average execution time of a packet calculated in step 1 of the algorithm. Once the average time is reached, a new stage is started. This process continues until all the elements are contained in a stage. Once this step is completed, we perform a local optimization stage where all the stage boundaries are considered. If moving an element from one stage to the

other (the move can be from stage i to i+1 or vice versa) reduces the overall variance in the total execution times of the stages, then the location of the element is changed. We traverse the stages until no element can be moved. This initial task distribution scheme is called the *Base Task Distribution (**BTD**)*. Figure 3.16 shows ***BTD*** scheme results on the *IPV4Router* application.

The statistical data obtained for each Click element shows on average approximately 30% of the data packets couldn't be processed within the mean processing time (μ). A slack of the form $k.\sigma$ in the estimated processing time helps a particular element to process a packet within the estimated delay. Therefore, instead of forming the stages using the mean processing time (μ), we form the stages using $\mu + k.\sigma$ as the expected execution time. In other words, the weight of each tree node (element) is set to $\mu + k.\sigma$. This scheme is called *Extended Task Distribution (**ETD**)*. The intuition behind ETD is to allow each element an extended slack to process a packet. By allowing an extended processing delay, we expect more packets to be processed within the expected processing time, improving the resource utilization in the Network Processor. We have performed a number of experiments with varying the k value. Our experiments reveal that the optimal point across the applications is achieved for $k = 3$. The detailed experiments are described in Section 3.12.

3.11.3 Selective Module Replication

It is normal to encounter a situation where the number of different modules available in an application is less than the available execution core. In such cases,

we replicate the modules to parallelize the packet processing for that particular module. This way, the number of total modules is increased and the task distribution can be performed more evenly. However, instead of a naive replication scheme, we select the modules with the highest mean processing times. This replication scheme is called *Selective Replication (**SR**)*. The intuition behind SR is to allow a module run faster if it is one of the slower (or longer) ones. Once we replicate a module, the two new modules are assigned a weight equal to the half of the weight of the original module. This replication is continued until we generate enough modules for the given number of processors.

The Click elements can perform a variety of unit task. It can do simple computations like calculating checksum of a packet. At the same time, a single element can be used to implement DRR scheduling [87] . As a result, the average processing time for each element varies over a long range. In a typical application, we can expect a module to contain elements, which are used in two alternate routes and one of them has a larger average processing time than the other. Under the SR technique, we would replicate both the elements in execution cores. Whenever a packet traverses a ROUTE involving the element with smaller processing delay, the execution core would sit idle for most of its time. This would reduce the utilization of the core that directly contradicts our objective for task allocation. To counter this problem, we propose the Extended Selective Replication (ESR). In ESR, we select the elements with longer average processing time in a module and replicate them over more than one execution core. The parallelization

of the longer module reduces the total processing time of the module. As a result, the utilization of the execution cores performing the module task increases. Additionally, we consider an extended slack version of the SER technique where we allow an extra slack of $k.\sigma$ for each module. We call this technique as the Extended Selective Replication (ESR).

3.11.4 Discussion

We must note that our overall algorithm is based on profiling information. In general, the success of a profiling scheme is largely dependent on the correct selection of the input data sets. However, our experience with the networking applications studied in this research work shows that they exhibit very similar behavior even with different input packet traces. Particularly, we have tested the four applications using three different sets of packets from the NLANR traces. For the three input sets, the mean execution times and the standard deviation for the four applications varied by less than 3%. On the other hand, our experiments shows that the behavior of the applications was very much dependant on the "internal" data structures. For example, a change in the routing table structures used in IPV4Router application has a significant impact on the mean execution time of a number of elements. Therefore, to achieve effective task distribution, a user needs to carefully select the internal structures that will represent the working conditions of an application.

3.12. Experiments

3.12.1 Experimental Setup

The SimpleScalar/ARM version 3.0 simulator [89] is used to evaluate the proposed techniques. We modified the processor configuration to model a processor similar to execution cores in a variety of NP architectures. We compiled the Click router to run in the user level mode. It is modified to run in collaboration with the SimpleScalar simulator. The SimpleScalar simulator is modified to record the behavior of every packet within a configuration. With the use of marker elements within the configuration, we track the every packet within a click configuration and record the performance of Click elements processing the packets. We simulate four representative networking applications as discussed in Section 3.9.

We perform two sets of experiments. First, we analyze the proposed task allocation scheme from the throughput perspective. Particularly, we measure the throughput for increasing number of processors. In the second set of experiments, we study the effectiveness of the proposed optimizations on the task allocation. We measure the resource (i.e., processor) utilization of the studied applications with BTD, ETD, SR, and ESR schemes.

3.12.2 Throughput Analysis

We have analyzed the system throughput under the task distribution and the replication schemes. For the processor with 2, 4, and 8 cores, we measure the average throughput of the system. Figure 3.17 through Figure 3.20 describe the relative throughput of the Multicore systems for the experimental applications.

The figures present the results for 4 variations as described in Section 3.11: Base Task Distribution (BTD), Extended Task Distribution (ETD), Selective Replication (SR), and Extended Selective Replication (ESR). Note that the distribution of stages in SR is based on BTD and ESR uses ETD strategy to place the modules into processor cores.

As we describe in Sections 3.11.2 and 3.11.3, ESR and ETD schemes use $\mu + k\sigma$ as the expected processing time. Therefore, we need to select a k value. Our tests with different k values showed that k=3 gives the best results overall. For small k values, the stages for ESR and ETD were identical to those of SR and BTD, respectively. For larger k values, on the other hand, the elements with large variance were assigned to single cores (e.g, stages formed by only such elements). If the execution time for a packet is close to or smaller than the mean processing time, this particular core is underutilized, reducing overall utilization. The optimal point is achieved when both the producer and the consumer are utilized fully. The selection of k=3 is the closest case to this scenario.

Figure 3.17 through Figure 3.20 present relative performance achieved with respect to single core execution of the original application after applying the proposed schemes. We see that task distribution is highly scalable for all the schemes. On average, for 2, 4, and 8 processors, BTD scheme achieves a relative throughput of 1.78, 3.25, and 6.15, respectively. The best throughput improvement is observed for DRR application. The reason for this behavior lies in the unique nature of this application. DRR contains two elements with large execution times.

Hence, we can achieve close to perfect task distribution for two processors. With SR schemes, the relative throughput for 2, 4, and 8 processors are 1.84, 3.40, and 6.42, respectively. For all the applications, we can see that SR performs better than BTD scheme. Due to replication of the processing elements that takes longer time, SR scheme improves the overall utilization of the processing cores. We can notice the improvement for higher number of cores (4 or 8) as that allows us for intelligent allocation of resources. The best performance is observed for the *IPv4Router* application. It has a relative throughput of 6.55 for 8 processors. As shown in Table 2, the *IPv4Router* application has a large variation of processing time for different elements. This variation gets benefited by the SR scheme to have even processing time for each pipelined stage and subsequently resulting high throughput scalability.

The extended version of BTD and SR includes an extra slack of 3σ to the expected processing time of the elements while task allocation. As shown in the figures, this results in a throughput improvement for almost all the cases. The extended schemes perform particularly well for the RED application. For ESR, the best performance is observed for the 8-core configuration, when the throughput reaches 7, a 12.5% improvement over BTD. The reason for this improvement lies in the nature of this application. RED consists of a number of elements with mean processing times close to each other. Therefore, by considering the standard deviation in the execution times, we see that the stage formations can be significantly changed.

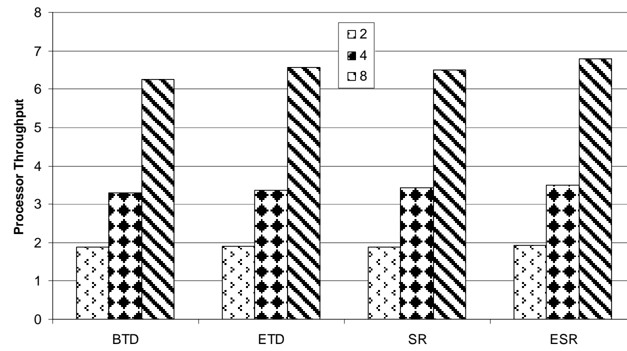


Figure 3.17. *Processor throughput for DRR application*

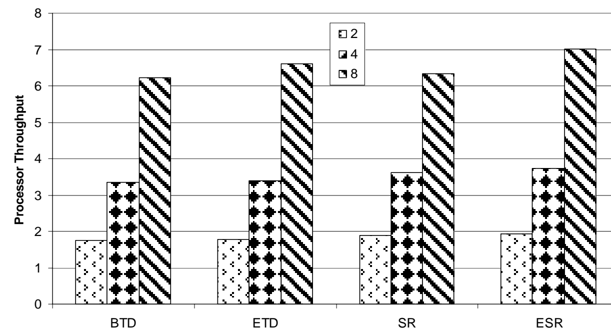


Figure 3.18. *Processor throughput for RED application*

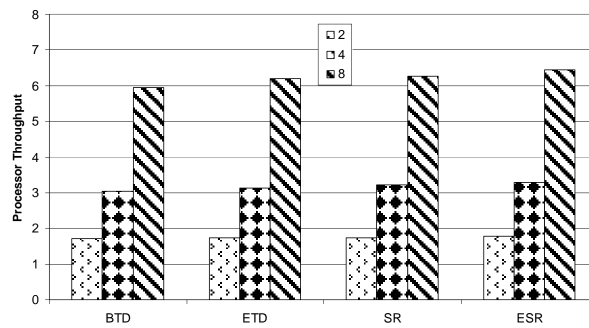


Figure 3.19. *Processor throughput for Home_Node application*

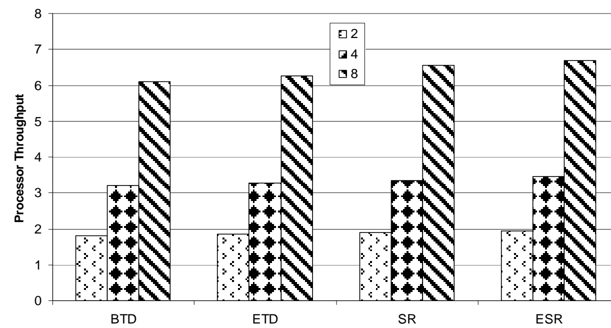


Figure 3.20. *Processor Throughput in Route application*

For other applications that are dominated by a few elements, consideration of the extended processing times usually does not cause a significant change in the stage formation. This is especially true for the DRR application, where the majority of the processing time is dominated by two modules. Moreover, ETD scheme always improves the throughput for 8 processor configurations. On average, for all four applications, it improves the throughput by 4.4%. On the other hand, the ESR scheme achieves a relative throughput of 1.90, 3.49, 6.74 for 2, 4, and 8 core processors, respectively, aggregated over all four applications. Henceforth, the combination of *Selective Replication* and *Extended Slack* results significant throughput improvement. On average, it improves the throughput by 6.4%, 8.4%, and 9.9% for 2, 4, and 8 processors as compared to the BTD scheme. We must note that the overall performance improvement achieved by our proposed schemes is synergistic. While the consideration of variance (ETD) and replication (SR) improve the performance by 4.4% and 4.3%, respectively, their combination (ESR) provides 9.9% improvement.

3.12.3 Resource Utilization Analysis

In the second set of experiments, we measure the average utilization of the cores. Figure 3.21 describes the mean utilization percentage of the cores for the DRR application. The figure presents the results for 4 variations as described in Section 5: the Base Task Distribution (BTD), the Extended Task Distribution (ETD), the Selective Replication (SR), and the Extended Selective Replication (ESR). The other applications follow the similar trends.

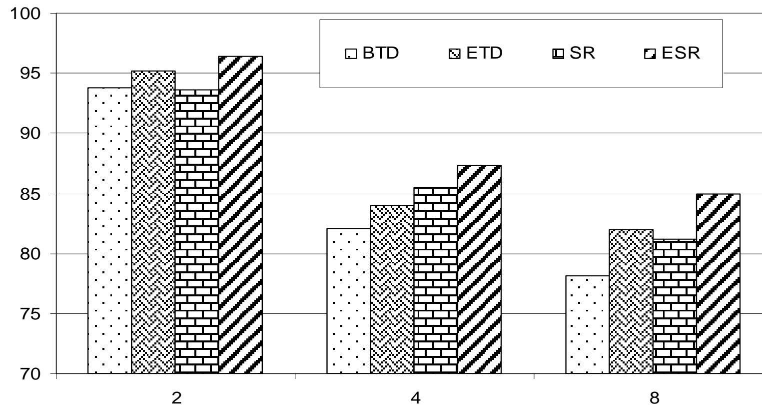


Figure 3.21. *Resource Utilization in DRR application*

For almost all the applications, we see that the ESR scheme gives the best utilization. In general, we see that both of our optimizations (replication and extended processing time consideration) increase the utilization. Particularly, SR almost always performs better than the BTD. A larger number of elements are useful for achieving a higher utilization. With the larger number of elements, we can form stages that are close to each other in the execution times. The only exception to this rule is the DRR application executed on 2 cores. For this application, SR and BTD provide the same throughput. The reason for this behavior lies in the unique property of the DRR application. DRR contains two elements with large execution times. Hence, without replication, we can achieve close to perfect task distribution. Particularly, DRR achieves the best utilization for 2-core processor with 95%, which cannot be improved with the SR scheme. In addition, we see that the ETD and ESR schemes always perform better than the BTD and SR schemes, respectively. We observe a similar trend for the remaining three applications. Overall, the ESR scheme achieves the best utilization with 95%,

89%, and 84% on average for the processors with 2, 4, and 8 cores, respectively, aggregated over all four applications. The average utilization for the BTD scheme, on the other hand, is 88%, 81%, and 77% for the processors with 2, 4, and 8 cores, respectively.

3.13. Related Work on Task Allocation

Task allocation has been an active research area in a number of domains. In behavioral synthesis research, the objective is to assign operations to hardware and optimize the usage of storage and communication paths [90]. While analogous to the problem faced here, these approaches are best suited for synthesizing datapath elements for small computational kernels. In the multiprocessor domain, Chekuri et al. [91] and Shachnai [92] proposed approximation algorithms to solve the problem for general multiprocessor models. However, they fail to consider practical resource constraints and do not take into account thread and storage limitations, which are critical factors that affect the quality of the mapping to heterogeneous ASIP architectures.

We have used Click infrastructure for our experimentation. It is the most relevant and established academic C++ programming model and environment for building packet processing applications on a single, general-purpose, processor. Shangri-La [93] is a work that matches our interest pretty closely. Shangri-La is based on Baker [75] that bears many similarities to Click, especially in regards to its modeling of communication channels (CCs). However, Their approach is significantly different than ours and we specifically concentrate on mapping the

tasks to cores. The optimizations presented in their scheme do not consider the variations observed in the packet execution. Due to architectural and technology differences, it is difficult to make any performance comparison between our system and theirs. Gordon et. al. [94] proposed schemes that employs task distribution among different cores similar to ours. The novelty of our scheme lies on the fact that the task distribution is based on the statistical properties of the network packets.

Plishker [95] exploited the flexible framework of ILP to generate optimal solutions to the mapping problem. Such techniques are usually computationally expensive. Srinivasan et al. [96] considered the scheduling problem for the Intel IXP1200 and presented a theoretical framework in order to provide service guarantees to applications. However, their methodology was not tested with real network applications.

A number of programming environments were proposed for NPs. NP-Click [74] is an extended programming model based on Click Router. It is implemented on the Intel IXP1200 architecture. Memik and Mangione-Smith [97] proposed a programming environment that considers the task allocation. However, none of these techniques used the variation in execution time to optimize the allocation schemes. Datar and Franklin [98] proposed greedy-pipe algorithm to solve problems associated with determining optimal application task assignments for pipelines in CMP based NP. However, their study is performance oriented and the execution core utilization has not done by them.

3.14. Conclusions

In this chapter, we have discussed the application of holistic architecture approach at the application level of abstraction. First, we proposed the design and utilization of clumsy packet processors. Clumsy packet processors use the robustness available in the networking applications to increase the efficiency of hardware structures while increasing their fault probabilities. Overall, this results in better execution efficiency and reduced energy consumption. Particularly, we have shown how the access delay and energy consumption of a data cache can be reduced while increasing the hardware faults during accesses. We developed a realistic model that estimates the fault probability of the cache for a given clock frequency. Thus, a clumsy processor can increase the clock frequency of its data cache and reduce its energy consumption. We have also defined various application-specific error metrics that is used to measure the "*fallibility*" of the processor. Particularly, we have proposed the energy-delay-fallibility product metric, which can be used to measure the trade off between the energy, execution time, and the error probability. We have presented a scheme to adapt the frequency of the data cache to adjust to the application requirements. Our simulations reveal that there is a significant gap between the specifications of the circuit designer and the optimal clock frequency in terms of energy-delay^2-fallibility^2 product.

We have also presented a method for allocating tasks in Network Processors. The task allocation scheme utilized the modular nature of networking

applications. Variation in execution time is an inherent property of processing. The goal is to estimate this variation for different parts of the code and perform the task allocation accordingly. Our scheme assigns a module to each execution core of the Network Processor. The variance in packet processing time is used to allow extra slack in each module. In addition, we present two schemes to replicate modules if the number of modules in the application is low. The first one (SR) simply replicates the modules based on their execution time, whereas the second one (ESR) considers the variation in execution time of the modules when making replication decisions. Results reveal several important characteristics of our proposed schemes. First, they show that the base task distribution scheme achieves high levels of scalability. In addition, the extended processing time and replication scheme help to improve the performance.

CHAPTER 4

User-Directed Power Management

The increasing importance of low-power VLSI design has resulted in numerous power-reduction techniques in circuits, architectures, and operating systems. Energy consumption has traditionally been one of the primary design criteria for mobile systems. It determines battery life and is therefore of great importance to end-users of mobile systems, a huge and growing population. In line-powered systems, on the other hand, energy consumption is important due to its impact on power dissipation, which affects cost and noise. As manufacturing technologies are enhanced, more and more transistors can be packed into a given area, increasing the power density. As a result, in high-end microprocessors, the chip temperature during execution is elevated, affecting performance, reliability, and integrated circuit (IC) lifetime.

Dynamic Voltage and Frequency Scaling (DVFS) is one of the most commonly used power reduction techniques in high-performance processors and is the most important OS power management tool. DVFS is generally implemented in the kernel and it varies the frequency and voltage of a microprocessor in real-

time according to processing needs. Although there are different versions of DVFS, at its core DVFS adapts power consumption and performance to the current workload of the CPU. Specifically, existing DVFS techniques in high-performance processors select an operating point (CPU frequency and voltage) based on the utilization of the processor. While this approach can integrate information available to the OS kernel, such control is pessimistic:

- Existing DVFS techniques are pessimistic about the user. Indeed, they ignore the user, assuming that CPU utilization or the OS events prompting it are sufficient proxies. A high CPU utilization simply leads to a high frequency and high voltage, regardless of the user's satisfaction or expectation of performance.

- Existing DVFS techniques are pessimistic about the CPU. They assume worst-case manufacturing process variation and operating temperature by basing their policies on loose worst-case bounds given by the processor manufacturer. A voltage level for each frequency is set such that even the slowest shipped processor of a given generation will be stable at the highest specified temperature.

In response to these observations, on which we elaborate in Sections 4.1 and 4.2, we have developed two new power management techniques that can be readily employed independently or together. This work is done in collaboration

with Bin Lin who is a graduate student interested in Systems research. In particular, we introduce the following techniques.

User-Driven Frequency Scaling (UDFS) uses direct user feedback to drive an online control algorithm that determines the processor frequency (Section 4.1). Processor frequency has strong effects on power consumption and temperature, both directly and also indirectly through the need for higher voltages at higher frequencies. The choice of frequency is directly visible to the end-user as it determines the performance he sees. There is considerable variation among users with respect to the satisfactory performance level for a given workload mix. UDFS exploits this variation to customize frequency control policies dynamically to the *individual* user. Unlike previous work (Section 4.5), this approach employs direct feedback from the user during ordinary use of the machine.

Process-Driven Voltage Scaling (PDVS) creates a custom mapping from frequency and temperature to the minimum voltage needed for CPU stability (Section 4.2), taking advantage of process variation. This mapping is then used online to choose the operating voltage by taking into account the current operating temperature and frequency. Researchers have shown that process variation causes IC speed to vary up to 30% [99]. Hence, using a single supply voltage setting does not exploit the variation in timing present among processors. We take advantage of this variation via a customization process that determines the slack of the *individual* processor, as well as its dependence on operating temperature. This

offline measurement is used online to dynamically set voltage based on frequency and temperature.

4.0.1 Experimental Setup

Our experiments were done using an IBM Thinkpad T43p with a 2.13 GHz Pentium M-770 CPU and 1 GB memory running Microsoft Windows XP Professional SP2. Although eight different frequency levels can be set on the Pentium M-770 processor, only six can be used due to limitations in the SpeedStep technology.

In all of our studies, we make use of three application tasks, some of which are CPU intensive and some of which frequently block while waiting for user input:

- Creating a presentation using Microsoft PowerPoint 2003 while listening to background music using Windows Media Player 10. The user duplicates a presentation consisting of complex diagrams involving drawing and labeling, starting from a hard copy of a sample presentation.

- Watching a 3D Shockwave animation using the Microsoft Internet Explorer web browser. The user watches the animation and is encouraged to press the number keys to change the camera's viewpoint. The animation was stored locally. Shockwave options were configured so that rendering was done entirely in software on the CPU.

- Playing the FIFA 2005 Soccer game. FIFA 2005 is a popular and widely-used First Person Shooter game. There were no constraints on user gameplay.

In the following sections, we describe the exact durations of these tasks for each user study and additional tasks the user was asked to undertake. In general, our user studies are double-blind, randomized, and intervention-based. The default Windows DVFS scheme is used as the control. We developed a user pool by advertising our studies within a private university that has many non-engineering departments. We selected a random group of users from among those who responded to our advertisement. While many of the selected users were CS, CE, or EE graduate students, our users included staff members and undergraduates from the humanities. Each user was paid $15 for participating. Our studies ranged from number of users n=8 to n=20, as described in the material below.

4.1. User-Driven Frequency Scaling

Current DVFS techniques are pessimistic about the user, which leads them to often use higher frequencies than necessary for satisfactory performance. In this section, we elaborate on this pessimism and then explain our response to it: user-driven frequency scaling (UDFS). Evaluations of UDFS algorithms are given in Section 4.3.1.

4.1.1 Pessimism about the user

Current software that drives DVFS is pessimistic about the individual user's reaction to the slowdown that may occur when CPU frequency is reduced.

Typically, the frequency is tightly tied to CPU usage. A burst of computation due to, for example, a mouse or keyboard event brings utilization quickly up to 100% and drives frequency, voltage, temperature, and power consumption up along with it. CPU-intensive applications also cause an almost instant increase in operating frequency and voltage.

In both cases, the CPU utilization (or OS events that drive it) is functioning as a proxy for user comfort. Is it a good proxy? To find out, we conducted a small (n=8) randomized user study, comparing four processor frequency strategies including dynamic, static low frequency (1.06 GHz), static medium frequency (1.33 GHz), as well as static high frequency (1.86 GHz). The dynamic strategy is the default DVFS policy used in Windows XP Professional. Note that the processor maximum frequency is 2.13 GHz. We allowed the users to acclimate to the full speed performance of the machine and its applications for 4 minutes and then carry out the tasks described in Section 4.0.1, with the following durations:

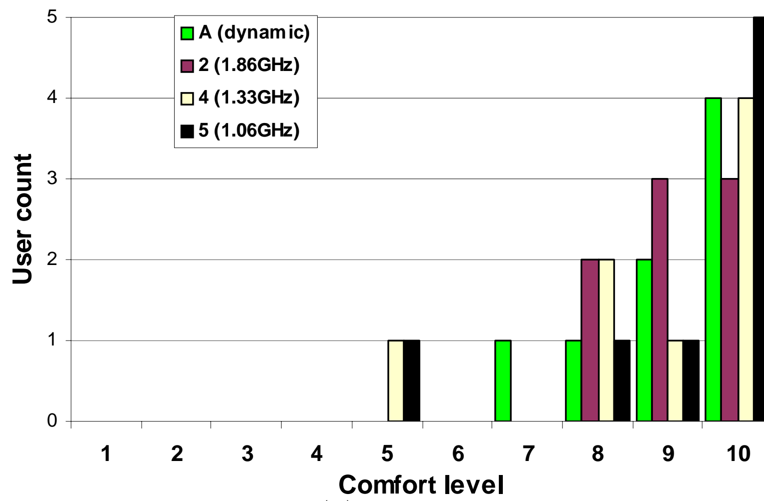
- PowerPoint (4 minutes in total, 1 minute per strategy)
- Shockwave (80 seconds in total, 20 seconds per strategy)
- FIFA (4 minutes in total, 1 minute per strategy)

Users verbally ranked their experiences after each task / strategy pair on a scale of 1 (discomfort) to 10 (very comfortable). Note that for each application and user, strategies were tested in random order.

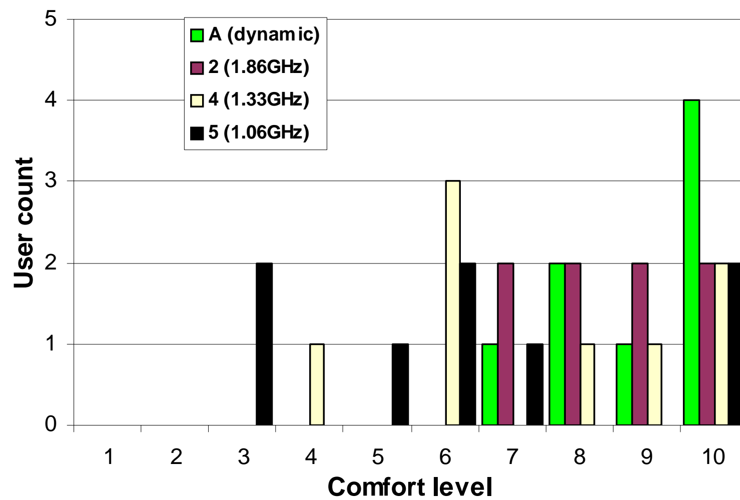
Figure 4.1 illustrates the results of the study in the form of overlapped histograms of the participants' reported comfort level for each of four strategies. Consider Figure 4.1(a), which shows results for the PowerPoint task. The horizontal axis displays the range of comfort levels allowed in the study and the vertical axis displays the count of the number of times that level was reported. The other graphs are similar.

User comfort with any given strategy is highly dependent on the application. For PowerPoint, the strategies are indistinguishable in their effectiveness. For this task, we could simply set the frequency statically to a very low value and never change it, presumably saving power. For animation, a higher static level is preferred but the medium and high frequencies are statistically indistinguishable from the dynamic strategy despite not using as high a frequency. For the game, the high static setting is needed to match the satisfaction level of the dynamic strategy. However, that setting does not use the highest possible frequency, which was used by the dynamic strategy throughout the experiment.

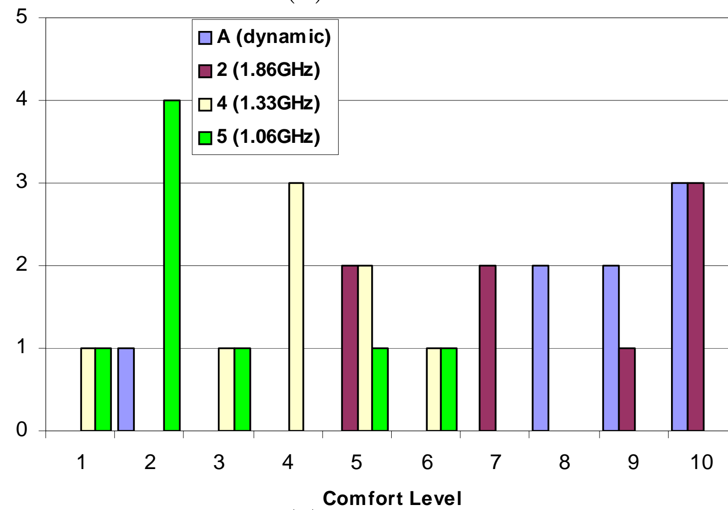
Comfort with a given strategy is strongly user-dependent, i.e., it is important to note that for any particular strategy, there is considerable spread in the reported comfort levels. In addition to the power-specific results just described, we note that Gupta et al. [100] and Lin et al. [101] have also demonstrated a high variation in user tolerance for performance in other contexts. Our dynamic policy automatically adapts to different users and applications. Hence, it can reduce power consumption while still achieving high user satisfaction.



(a) PowerPoint



(b) 3D Animation



(c) FIFA Game

Figure 4.1. *User pessimism.*

4.1.2 Technique

To implement user-driven frequency scaling, we have built a system that consists of client software that runs as a Windows toolbar task as well as software that implements CPU frequency and temperature monitoring. In the client, the user can express discomfort at any time by pressing the F11 key (the use of other keys or controls can be configured). These events drive the UDFS algorithm. The algorithm in turn uses the Windows API to control CPU frequency. We monitor the CPU frequency using Windows Performance Counter and Log [102] and temperature using CPUCool [103].

It is important to note that a simple strategy that selects a static frequency for an application (and/or for a user) is inadequate for three reasons. First, each user will be satisfied with a different level of performance for each application. Finding these levels statically would be extremely time consuming. Second, typical users multitask. Capturing the effects of multiple applications would necessitate examining the power set of the application set for each individual user, resulting in a combinatoric explosion in the offline work to be done. Finally, even when a user is working with a single application, the behavior of the application and the expected performance varies over time. Applications go through phases, each with potentially different computational requirements. In addition, the user's expected performance is also likely to change over time as the user's priorities shift. For these reasons, a frequency scaling algorithm should dynamically adjust to the individual user's needs.

Responding to these observations, we designed algorithms that employ user experience feedback indicated via button presses.

4..1.2.1 UDFS1 Algorithm

UDFS1 is an adaptive algorithm that can be viewed as an extension/variant of the TCP congestion control algorithm. The TCP congestion control algorithm [104-106] is designed to adapt the send rate dynamically to the available bandwidth in the path. A congestion event corresponds to a user button press, send rate corresponds (inversely) to CPU frequency, and TCP acknowledgments correspond to the passage of time.

UDFS1 has two state variables: r, the current control value (CPU frequency, the smaller the value, the higher the frequency.) and r_t (the current threshold, integer value). Adaptation is controlled by three constant parameters: ρ, the rate of increase, $\alpha=f(\rho)$, the slow start speed, and $\beta=g(\rho)$, the additive increase speed. Like TCP, UDFS1 operates in three modes, as described below.

Slow Start (Exponential Increase): If $r < r_t$, we increase r exponentially fast with time (e.g. $r \propto 2^{\alpha t}$). Note that frequency settings for most processors are quantized and thus the actual frequency changes abruptly upon crossing quantization levels.

User event avoidance (Additive Increase): If no user feedback is received and $r \geq r_t$, r increases linearly with time, $r \propto \beta t$.

User event (Multiplicative Decrease): When the user expresses discomfort at level r we immediately set r_t=r_t-1 and set r to the initial (highest) frequency.

This behavior is virtually identical to that of TCP Reno, except for the more aggressive setting of the threshold.

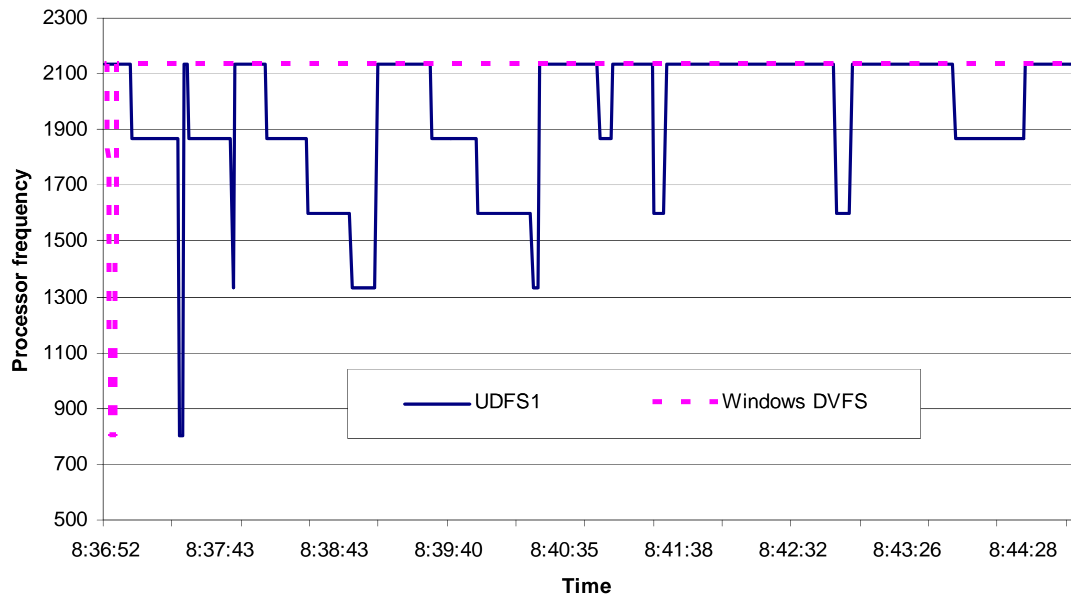
Unlike TCP Reno[106] , we also control ρ, the key parameter that controls the rate of exponential and linear increase from button press to button press. In particular, for every user event, we update as follows:

$$\rho_{i+1} = \rho_i (1 + \gamma \times \frac{T_i - T_{AVI}}{T_{AVI}})$$

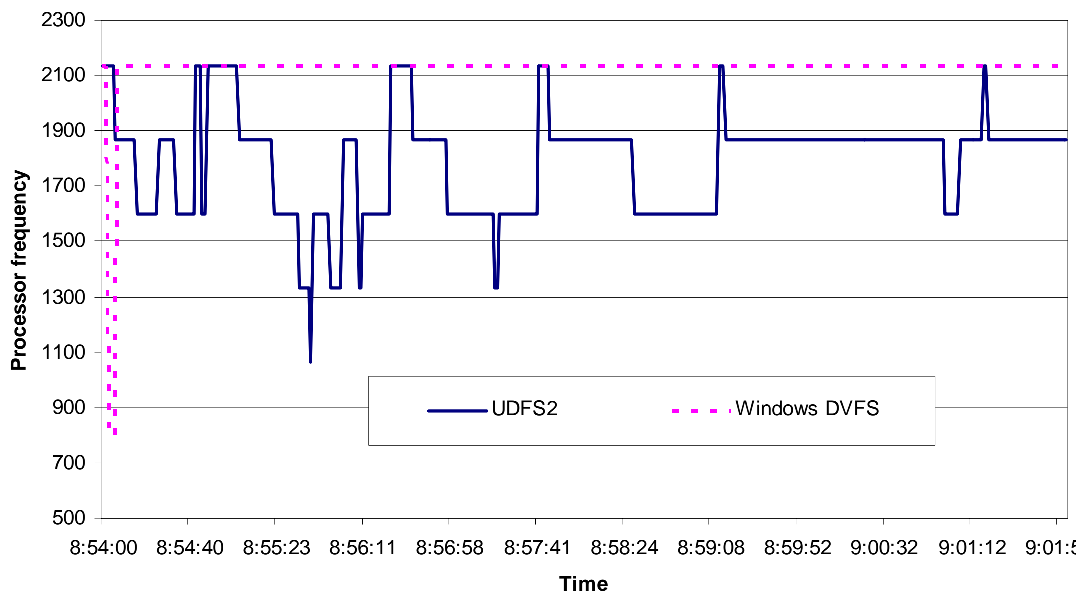
where T_i is the latest inter-arrival time between user events. T_{AVI} is the target mean inter-arrival time between user events, as currently preset by us. controls the sensitivity to the feedback.

We set our constant parameters (T_{AVI}=120, α=1.5, β=0.8, γ=1.5) based on the experience of two of the authors using the system. These parameters were subsequently used when conducting a user study to evaluate the system (Section 4.3). Ideally, we would empirically evaluate the sensitivity of UDFS1 performance to these parameters. However, it is important to note that any such study would require having real users in the loop, and thus would be quite slow. Testing five values of each parameter on 20 users would require 312 days (based on 8 users/day and 45 minutes/user). For this reason, we decided to choose the

parameters based on qualitative evaluation by the authors and then “close the loop” by evaluating the whole system with the choices.



(a) UDFS1 Frequency Traces



(a) UDFS2 Frequency Traces

Figure 4.2. *The frequency for UDFS schemes during FIFA game for a representative user.*

Figure 4.2(a) illustrates the execution of the UDFS1 and Windows DVFS algorithms for a typical user during the FIFA game task. Note that Windows DVFS causes the system to run at the highest frequency during the whole execution period except the first few seconds. On the other hand, the UDFS1 scheme causes the processor frequency to increase only when the user expresses discomfort (by pressing F11). Otherwise, it slowly decreases.

4..1.2.2 UDFS2 Algorithm

UDFS2 tries to find the lowest frequency at which the user feels comfortable and then stabilize there. For each frequency level possible in the processor, we assign an interval t_i, the time for the algorithm to stay at that level. If no user feedback is received during the interval, the algorithm reduces the frequency from r_i to r_{i+1}, i.e., it reduces the frequency by one level. The default interval is 10 seconds for all levels. If the user is irritated at control level r_i, we update all of our intervals and the current frequency level as follows:

$$t_{i-1} = \alpha t_{i-1}$$

$$t_k = \beta t_k, \forall k : k \neq i-1$$

$$i = \min(i-1,0)$$

Here $\alpha > 1$ is the rate of interval increase and $\beta < 1$ is rate of interval decrease. In our study, $\alpha = 2.5$ and $\beta = 0.8$. This strategy is motivated by the conjecture that the user was comfortable with the previous level and the algorithm should spend more time at that level. Again, because users would have to be in the

inner loop of any sensitivity study, we have chosen the parameters qualitatively and evaluated the whole system using that choice, as described in Section 4.3.1. Figure 4.2(b) illustrates the execution of the algorithm for a representative user in the FIFA game task. Note that UDFS2 settles to a frequency of approximately 1.86GHz, after which little interaction is needed.

4.2. Process-Driven Voltage Scaling

Current DVFS techniques are pessimistic about the processor, which leads them to often use higher voltages than necessary for stable operation, especially when they have low temperatures. We elaborate on this pessimism and then explain our response to it, process-driven voltage scaling (PDVS). PDVS is evaluated in Section 4.3.2.

4.2.1 Pessimism about the CPU

The *minimum stable voltage* of a CPU is the supply voltage that guarantees correct execution for given process variation and environmental conditions. It is mainly determined by the critical path delay of a circuit. This delay consists of two components: transistor gate delay and wire delay. Gate delay is inversely related to the operating voltages used in the critical paths of the circuit. Furthermore, temperature affects the delay. In current technologies, carrier mobility in MOS transistors decreases with increasing temperature. This causes the circuits to slow down with increasing temperature. Wire delay is also temperature-dependent and increases under higher current/temperature conditions. The

maximum operating frequency (F_{max}) varies in direct proportion to the sustained voltage level in the critical timing paths, and inversely with temperature-dependent RC delay [107].

In addition to the operating conditions, which dynamically change, process variation has an important impact on the minimum voltage sufficient for stable operation. Even in identical environments, a variation in timing slack is observed among the manufactured processors of the same family. As a result, each processor reacts differently to changes. For example, although two processors can run safely at 2.8 GHz at the default supply voltage, it is conceivable that these minimum supply voltages will differ. Customizing voltage choices for individual processors adapts to, and exploits, these variations. Despite these known effects of process variation and temperature on minimum stable voltage, DVFS ignores them: for a given frequency, traditional DVFS schemes use a single voltage level for all the processors within a family at all times.

The dynamic power consumption of a processor is directly related to frequency and supply voltage and can be expressed using the formula $P = V^2CF$, which states that power is equal to the product of voltage squared, capacitance, and frequency. In addition to its direct impact on the power consumption, reliable operation at increased frequency demands increased supply voltage, thereby having an indirect impact on power consumption. Generally, if the frequency is reduced, a lower voltage is safe.

As processors, memories, and application-specific integrated circuits (ASICs) are pushed to higher performance levels and higher transistor densities, processor thermal management is quickly becoming a first-order design concern. The maximum operating temperature of an Intel Pentium Mobile processor has been specified as $100^{\circ}C$ [108, 109]. As a general rule of thumb, the operating temperature of a processor can vary from $50^{\circ}C$ to $90^{\circ}C$ during normal operation. Thus, there is a large difference between normal and worst-case temperatures.

We performed an experiment that reveals the relationship between operating frequency and minimum stable voltage of the processor at different temperature ranges. We used Notebook Hardware Control (NHC) [110] to set a particular V_{dd} value for each operating frequency supported by the processor. When a new voltage value is set, NHC runs an extensive CPU stability check. Upon failure, the system stops responding and computer needs to be rebooted. We execute a program that causes high CPU utilization and raises the temperature of the processor. When the temperature reaches a desired range, we perform the CPU stability check for a particular frequency at a user-defined voltage value.

Table 4-A shows the results of this study for the machine described in Section 4.0.1. For reference, we also show the nominal core voltage given in the datasheet [108]. Note that the nominal voltage is the voltage used by all the DVFS schemes by default. The results reveal that, even at the highest operating temperature, the minimum stable voltage is far smaller than the nominal voltage.

The results also show that at lower operating frequencies, the effect of temperature on minimum stable voltage is not pronounced. However, temperature change has a significant impact on minimum stable voltage at higher frequencies. In particular, at 1.46 GHz, the core voltage value can vary by 5.6% for a temperature change of 30°C. This would reduce dynamic power consumption by 11.4%.

Table 4-A. *Minimum stable V_{dd} for different operating frequencies and temperatures*

Operating Freq. (MHz)	Nominal Voltage (v)	Stable V_{dd} (V) at temp ranges (°C)			
		52–57	62–67	72–77	82–87
800	0.988	0.736	0.736	0.736	0.736
1,060	1.068	0.780	0.780	0.780	0.780
1,200	1.100	0.796	0.796	0.796	0.796
1,330	1.132	0.844	0.844	0.860	0.876
1,460	1.180	0.876	0.892	0.908	0.924
1,600	1.260	0.908	0.924	0.924	0.924
1,860	1.324	1.004	1.004	1.020	1.020
2,130	1.404	1.084	1.100	1.116	1.116

As the results shown in Table 4-A illustrate, there is an opportunity for power reduction if we exploit the relationship between frequency, temperature, and the minimum stable voltage. The nominal supply voltage specified in the processor datasheet has a large safety margin over the minimum stable voltages. This is not surprising: worst-case assumptions were unnecessarily made at a number of design stages, e.g., about temperature. Conventional DVFS schemes are therefore pessimistic about particular *individual* CPUs, often choosing higher voltages than are needed to operate safely. They also neglect the effect of temperature, losing the opportunity to save further power.

4.2.2 Technique

We have developed a methodology for exploiting the process variation described in Section 4.2.1 that can be used to make *any* voltage and frequency scaling algorithm adapt to individual CPUs and their temperature, thereby permitting a reduction in power consumption.

Our technique uses offline profiling of the processor to find the minimum stable voltages for different combinations of temperature and frequency. Online temperature and frequency monitoring is then used to set the voltage according to the profile. The offline profiling is virtually identical to that of Section 4.2.1 and needs to be done only once. Currently, it is implemented as a watchdog timer-driven script on a modified Knoppix Live CD that writes the profile to a USB flash drive. To apply our scheme, the temperature is read from the online sensors that exist in the processor. The frequency, on the other hand, is determined by the dynamic frequency scaling algorithm in use. By setting the voltage based on the processor temperature, frequency, and profile, we adapt to the operating environment. While the frequency can be readily determined (or controlled), temperature changes dynamically. Hence, the algorithm has built-in filtering and headroom to account for this fact. Our algorithm behaves conservatively and sets the voltage such that even if there is a change of 5°C in temperature before the next reading (one Hertz rate), the processor will continue working correctly.

A reader may at this point be concerned that our reduction of the timing safety margin from datasheet norms might increase the frequency of timing errors.

However, PDVS carefully determines the voltage required for reliable operation for each processor; that is, it finds the *individual* processor's safety margin. Moreover, it decreases the operating temperature of the processor, which reduces the rates of lifetime failure processes. If characteristics of processors change as a result of wear, PDVS can adapt by infrequently, e.g., every six months, repeating the offline characterization process. To determine processor reliability when using reduced operating voltage, we ran demanding programs test the stability of different processor components, e.g., the ALU, at lower voltages. We have set the processor to work at modified supply voltages as indicated in Table 4-A. The system remained stable for approximately two months, at which point we terminated testing. Although observing the stable operation of one machine does not prove reliability, it is strong evidence.

4.3. Evaluation

We now evaluate UDFS and PDVS in isolation and together. We compare against the native Windows XP DVFS scheme, displaying reductions in power and temperature.

Our evaluations are based on user studies, as described in Section and elaborated upon here. For studies not involving UDFS, we trace the user's activity on the system as he uses the applications and monitor the selections DVFS makes in response. For studies involving UDFS, the UDFS algorithm is used online to control the clock frequency in response to user button presses. We begin by describing a user study of UDFS that provides both independent results and traces

for later use. Next, we consider PDVS as applied to the Windows DVFS algorithm. We then consider UDFS with and without PDVS, comparing to Windows DVFS. Here, we examine both dynamic CPU power (using simulation driven from the user traces) and system power measurement (again for a system driven from the user traces). In measurement, we consider not only power consumption, but also CPU temperature. Finally, we discuss a range of other aspects of the evaluation of the system.

The following claims are supported by our results:

- UDFS effectively employs user feedback to customize processor frequency to the individual user. This typically leads to significant power savings compared to existing dynamic frequency schemes that rely only on CPU utilization as feedback. The amount of feedback from the user is infrequent, and declines quickly over time as an application or set of applications is used.

- PDVS can be easily incorporated into any existing DVFS scheme, such as the default Windows scheme, and leads to dramatic reductions in power use by lowering voltage levels while maintaining processor stability.

- In most of the cases, the effects of PDVS and UDFS are synergistic: the power reduction of UDFS+PDVS is more than the sum of its parts.

- Multitasking increases the effectiveness of UDFS+PDVS.

- Together and separately, PDVS and UDFS typically decrease CPU temperature, often by large amounts, increasing both reliability and longevity. In addition, the effects of PDVS and UDFS on temperature are synergistic.

4.3.1 UDFS

To evaluate the UDFS schemes, we ran a study with 20 users. Experiments were conducted as described in Section 4.0.1. Each user spent 45 minutes to

- Fill out a questionnaire stating level of experience in the use of PCs, Windows, Microsoft PowerPoint, music, 3D animation video, and FIFA 2005 (2 minutes) from among the following set: “Power User”, “Typical User”, or “Beginner”;

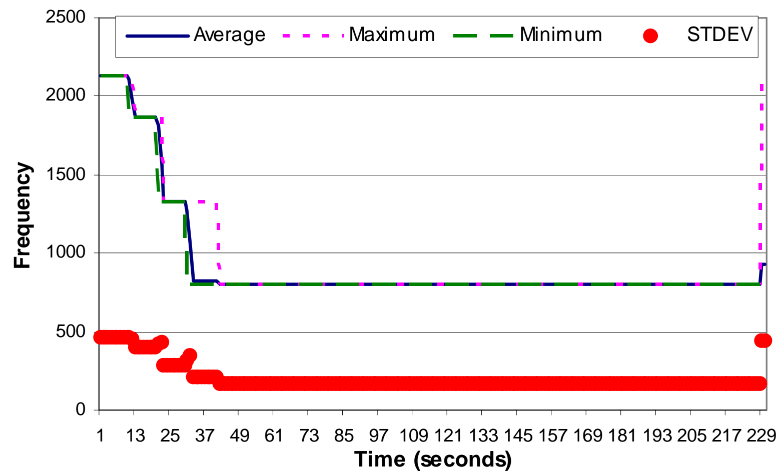
- Read a one page handout (2 minutes);

- Acclimate to the performance of our machine by using the above applications (5 minutes);

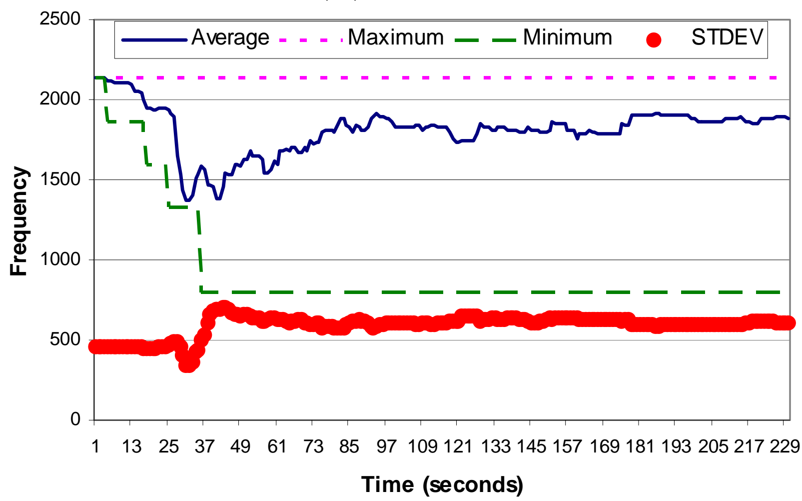
- Perform the following tasks for UDFS1: Microsoft PowerPoint plus music (4 minutes); 3D Shockwave animation (4 minutes); FIFA game (8 minutes); and

- Perform the same set of tasks for UDFS2.

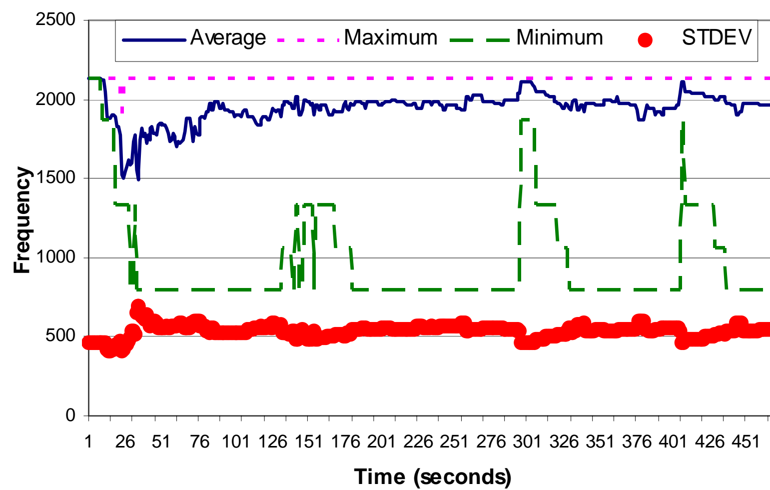
Each user was instructed to press the F11 key upon discomfort with application performance. We recorded each such event as well as the CPU frequency over time.



(a) PowerPoint

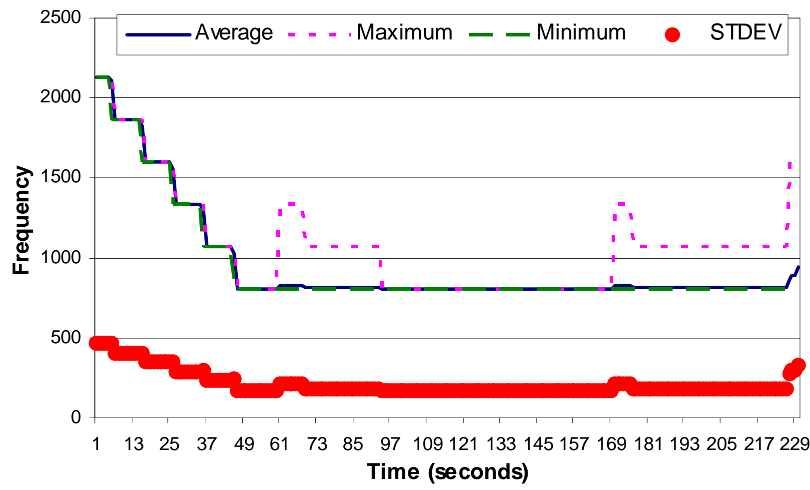


(b) 3D Animation

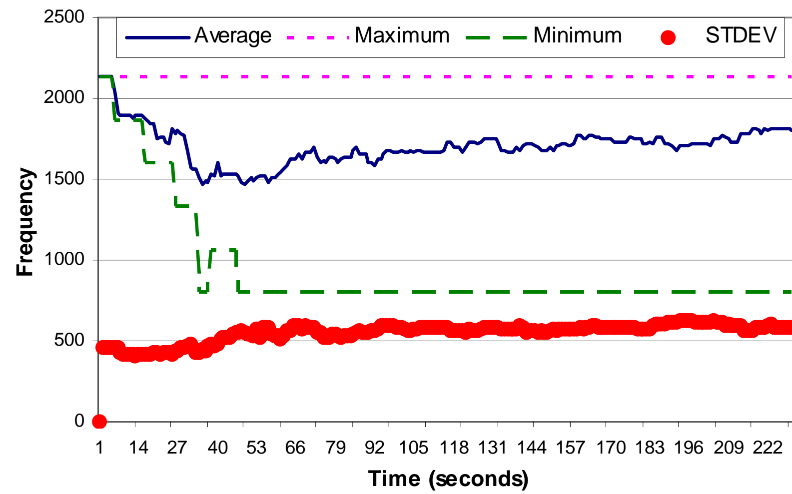


(c) FIFA Game

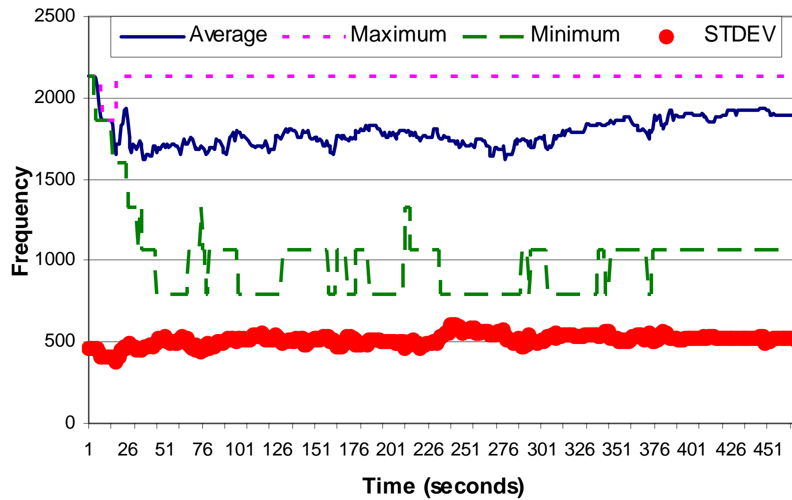
Figure 4.3. *Frequency over time for UDFS1 aggregated over 20 users.*



(a) PowerPoint



(b) 3D Animation



(c) FIFA Game

Figure 4.4. *Frequency over time for UDFS2 aggregated over 20 users.*

Figure 4.3 and Figure 4.4 illustrate the performance of the two algorithms in our study. The two columns represent UDFS1 and UDFS2 and the three rows represent the three applications. Each graph shows, as a function of time, the minimum, average, maximum, and standard deviation of CPU frequency, aggregated over our 20 users. Notice that almost all users felt comfortable using PowerPoint while the processor was running at the lowest frequency. As one might expect, the average frequency at which users are comfortable is higher for the Shockwave animation and the FIFA game. There is large variation in acceptable frequency among the users for the animation and game. Generally, UDFS2 achieves a lower average frequency than UDFS1. For both algorithms it is very rare to see the processor run at the maximum CPU frequency for these applications. Even the most sophisticated users were comfortable with running the tasks with lower frequencies than those selected by the dynamic Windows DVFS scheme. Sections 4.3.3 and 4.3.4 give detailed, per-user results for UDFS (and UDFS+PDVS).

4.3.2 PDVS

Using the experimental setup described in Section , we evaluate the effects of PDVS on the default Windows XP DVFS scheme. In particular, we run the DVFS scheme, recording frequency, then determine the power saving possible by setting voltages according to PDVS instead of using the nominal voltages of DVFS.

Table 4-B. *Power reduction for Windows DVFS and DVFS+PDVS*

Application	Power Reduction (%) over Max Freq.	
	DVFS	DVFS+PDVS
PowerPoint + Music	83.08	90.67
3D Shockwave Animation	3.19	40.67
FIFA Game	1.69	39.69

Table 4-B illustrates the average results, comparing stock Windows DVFS and our DVFS+PDVS scheme. The baseline case in this experiment is running the system with the highest possible CPU frequency and its corresponding nominal voltage. The maximum power savings due to dynamic frequency scaling with nominal voltages are observed for PowerPoint. For this application, the system ran at the lowest clock frequency most of the time, resulting in a reduction of 83.1% for the native DVFS scheme. DVFS+PDVS reduces the power consumption by 90.7%. For PowerPoint, adding PDVS to DVFS only reduces power slightly.

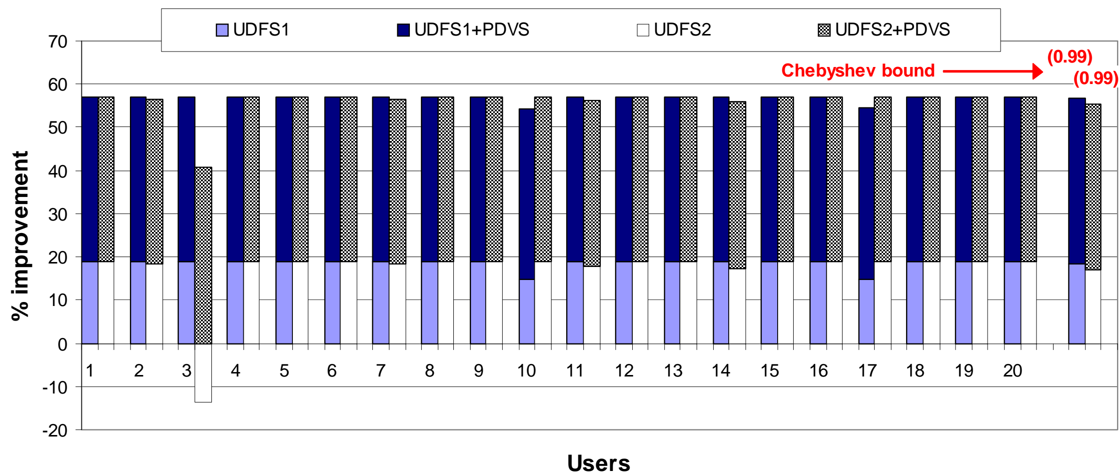
For the Shockwave animation and the FIFA game, the power reductions due to dynamic frequency scaling are negligible because the Windows DVFS scheme runs the processor at the highest frequency most of the time. DVFS+PDVS, however, improves the energy consumption of the system by approximately 40%, compared to the baseline. These results clearly demonstrate the benefits of process-driven voltage scaling.

4.3.3 UDFS+PDVS (CPU dynamic power, trace-driven simulation)

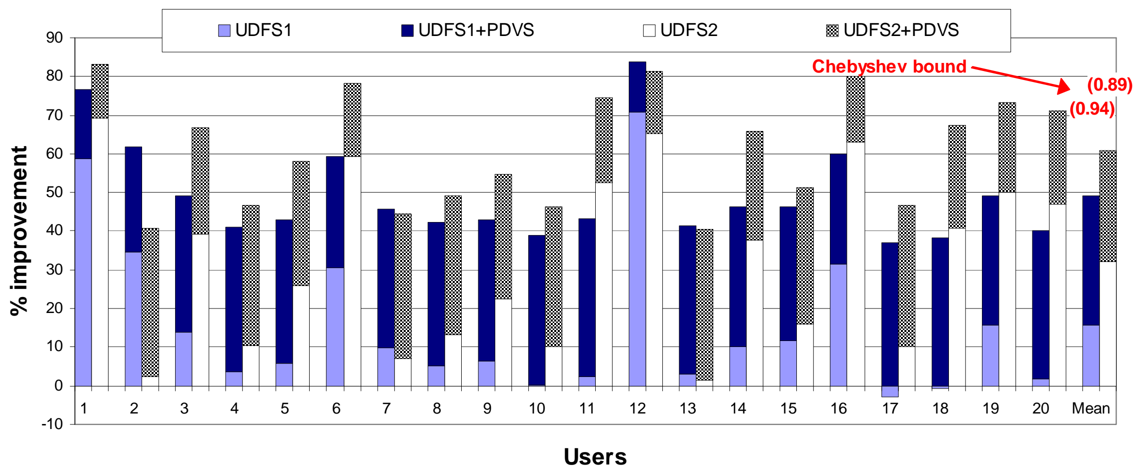
To integrate UDFS and PDVS, we used the system described in Section 4.1.2 recording frequency over time. We then combine this frequency

information with the offline profile and techniques described in Section 4.1.2 and 4.2.2 to derive CPU power savings for UDFS with nominal voltages, UDFS+PDVS, and the default Windows XP DVFS strategy. We calculate the power consumption of the processor. We have also measured online the power consumption of the overall system, as described in Section 4.3.4.

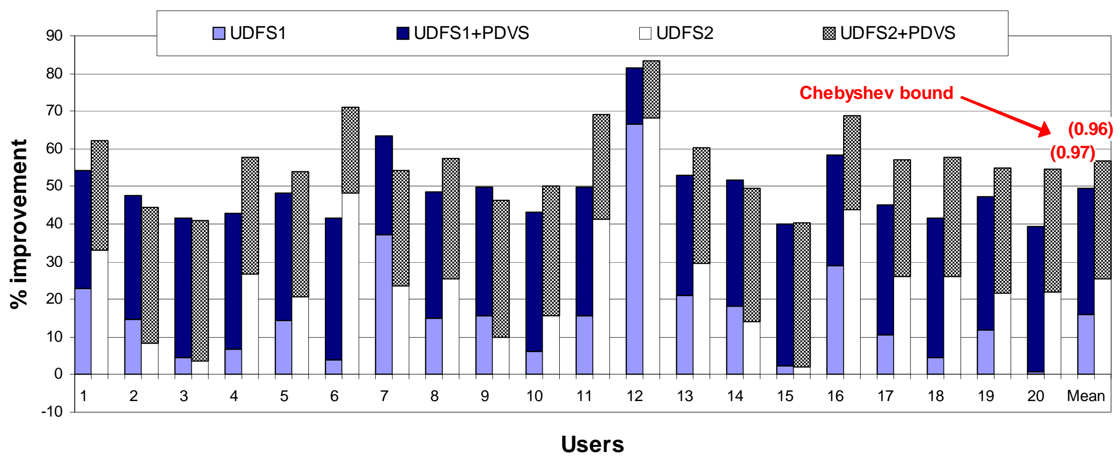
We conducted a user study (n=20) with exactly the same structure presented in Section 4.3.1, except that Windows XP DVFS was also considered. Figure 4.5 presents both individual user results and average results for UDFS1, UDFS1+PDVS, UDFS2, and UDFS2+PDVS. In each case, power savings over the default Windows DVFS approach are reported. To interpret the figure, first choose an application. Next, note the last two bars on the corresponding graph. These indicate the average performance of UDFS1 and UDFS2, meaning the percentage reduction in power use compared to Windows DVFS. Each bar is broken into two components: the performance of the UDFS algorithm without PDVS is the lower component and the improvement in performance of the algorithm combined with PDVS is the upper component. The remaining bars on the graph have identical semantics, but represent user-specific information.



(a) PowerPoint



(b) 3D Animation



(c) FIFA Game

Figure 4.5. *Comparison of UDFS algorithms, UDFS+PDVS, and Windows XP DVFS (CPU Dynamic Power). Chebyshev bound-based (1-p) values for difference of means from zero are also shown.*

For PowerPoint, UDFS1+PDVS and UDFS2+PDVS reduce power consumption by an average of 56%. The standalone UDFS algorithms reduce it by an average of 17–19%. User 3 with UDFS2 is anomalous. This user pressed the feedback button several times and as a result spent most of the time at high frequencies.

For the Shockwave animation, we see much more mixed responses from the users, although on average we reduce power by 55.1%. On average, UDFS1 and UDFS2 independently reduce the power consumption by 15.6% and 32.2%, respectively. UDFS2 performs better for this application because the users can be satisfied by ramping up to a higher frequency rather than the maximum frequency supported by the processor. Note that UDFS1 immediately moves to the maximum frequency on a button press. User 17 with UDFS1 is anomalous. This user wanted the system to perform better than the hardware permitted and thus pressed the button virtually continuously even when it was running at the highest frequency. Adding PDVS lowers average power consumption even more significantly. On average, the power is reduced by 49.2% (UDFS1+PDVS) and 61.0% (UDFS2+PDVS) in the combined scheme.

There is also considerable variation among users for the FIFA game. Using conventional DVFS, the system always runs at the highest frequency. The UDFS schemes try to throttle down the frequency over the time. They therefore reduce the power consumption even in the worst case (0.9% and 2.1% for UDFS1 and UDFS2, respectively) while achieving better improvement, on average (16.1% and

25.5%, respectively). Adding PDVS improves the average power savings to 49.5% and 56.7% for UDFS1 and UDFS2, respectively. For the Shockwave animation and the FIFA game, we see a large variation among users, but in all cases the combination of PDVS and UDFS leads to power savings over Windows DVFS. On average, in the best case, the power consumption can be reduced by 57.3% over existing DVFS schemes for all three applications. This improvement is achieved by combining the UDFS2 (24.9%) and PDVS (32.4%) schemes.

UDFS and PDVS are synergistic. The UDFS algorithms let us dramatically decrease the average frequency, and PDVS's benefits increase as the frequency is lowered. At higher frequencies, the relative change from the nominal voltage to the minimum stable voltage is lower than that at lower frequencies. In other words, the power gain from shifting to the minimum stable voltage is higher at the lower frequencies. However, at higher frequencies, PDVS also gains from the variation in minimum stable voltage based on temperature as shown in Table 4-A. These two different advantages of the PDVS result in power improvements at a wide range of frequencies.

UDFS+PDVS mean results have statistical significance even with weak bounds. Figure 4.5 shows mean improvements across our 20 users. Normality assumptions hold neither for the distribution of individual user improvements nor for the error distribution of the mean. Instead, to discard the null hypothesis, that our mean improvements for UDFS+PDVS are not different from zero, we have computed the p value for discarding the null hypothesis using Chebyshev bounds,

which are looser but rely on no such assumptions. As can be seen from the figure, 1-p is quite high, indicating that it is extremely unlikely that our mean improvements are due to chance. We use Chebyshev bounds similarly for other results. User self-reported level of experience correlates with power improvement. For example, for FIFA, experienced users expect faster response from the system causing the system to run at higher frequencies, resulting in smaller power improvements. Our interpretation is that familiarity increases both expectations and the rate of user feedback to the control agent, making annoyance with reduced performance more probable and thus leading to higher frequencies when using the UDFS algorithms.

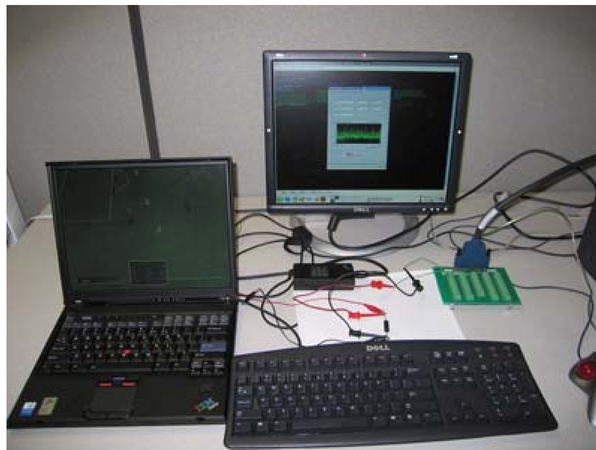


Figure 4.6. *System Power Measurement Setup*

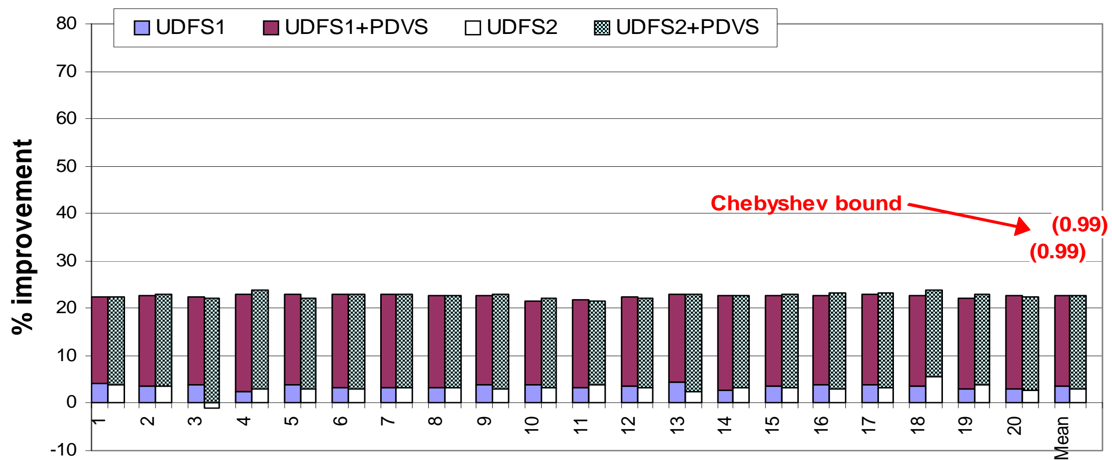
4.3.4 UDFS+PDVS (System power and temperature measurement)

To further measure the impact of our techniques, we replay the traces from the user study of the previous section on our laptop. The laptop is connected to a National Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux, which permits us to measure the power

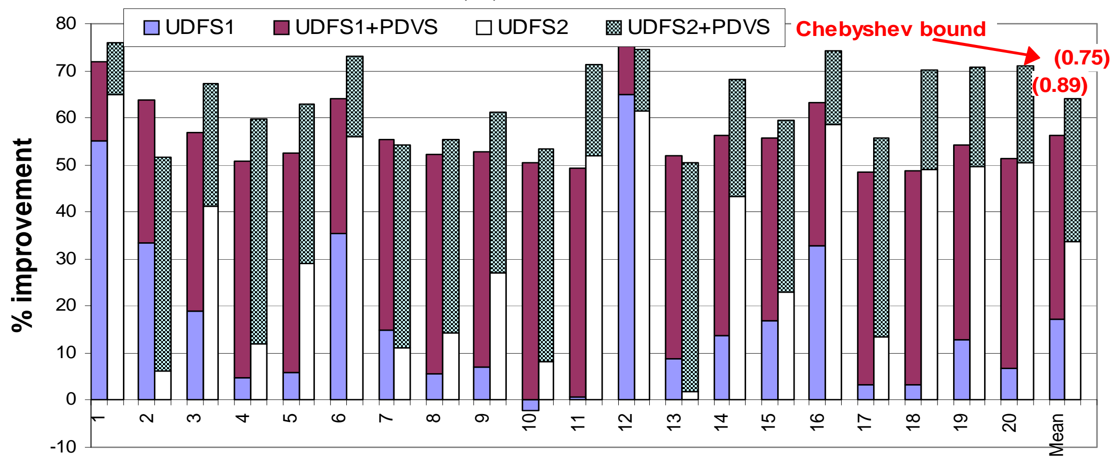
consumption of the entire laptop. The sampling rate is 10Hz. During the measurements, we have turned off the display of the laptop to make our readings more comparable to the CPU power consumption results of the previous section. Ideally, we would have preferred to measure CPU power directly for one-to-one comparison with results of the previous section, but we do not have the surface mount rework equipment needed to do so. Figure 4.6 shows the experimental setup used to measure the actual system power consumption.

4..3.4.1 Power

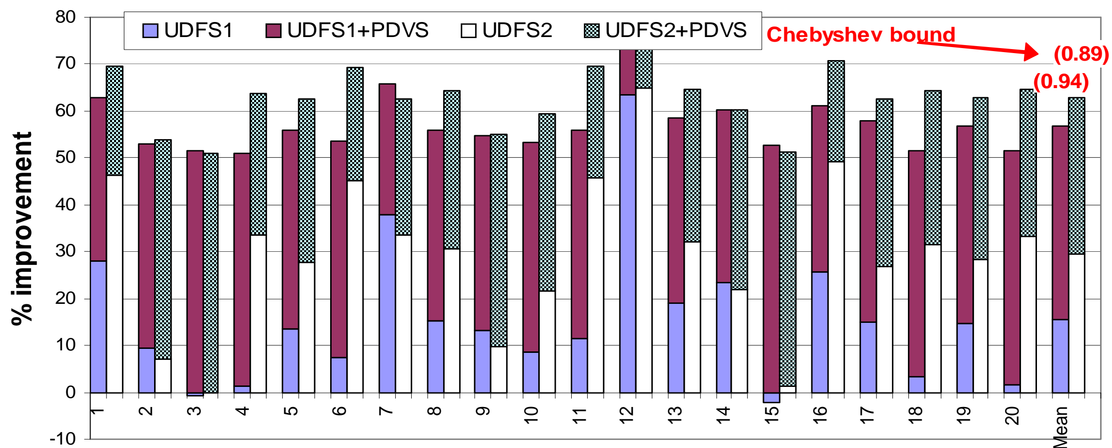
Figure 4.7 presents results for UDFS1, UDFS1+PDVS, UDFS2, and UDFS2+PDVS, showing the power savings over the default Windows DVFS approach. The Chebyshev bounds indicate that the mean improvements are extremely unlikely to have occurred by chance. For PowerPoint, UDFS1+PDVS and UDFS2+PDVS reduce power consumption by averages of 22.6% and 22.7%, respectively. For the Shockwave animation, although we see much more variation, UDFS1 and UDFS2 reduce the power consumption by 17.2% and 33.6%, respectively. Using UDFS together with PDVS lowers average power consumption by 38.8% and 30.4% with UDFS1 and UDFS2, respectively. The FIFA game also shows considerable variation among users. On average, we save 15.5% and 29.5% of the power consumption for UDFS1 and UDFS2, respectively. Adding PDVS improves the average power savings to 56.8% and 62.9% over Windows DVFS with UDFS1 and UDFS2, respectively.



(a) PowerPoint



(b) 3D Animation



(c) FIFA Game

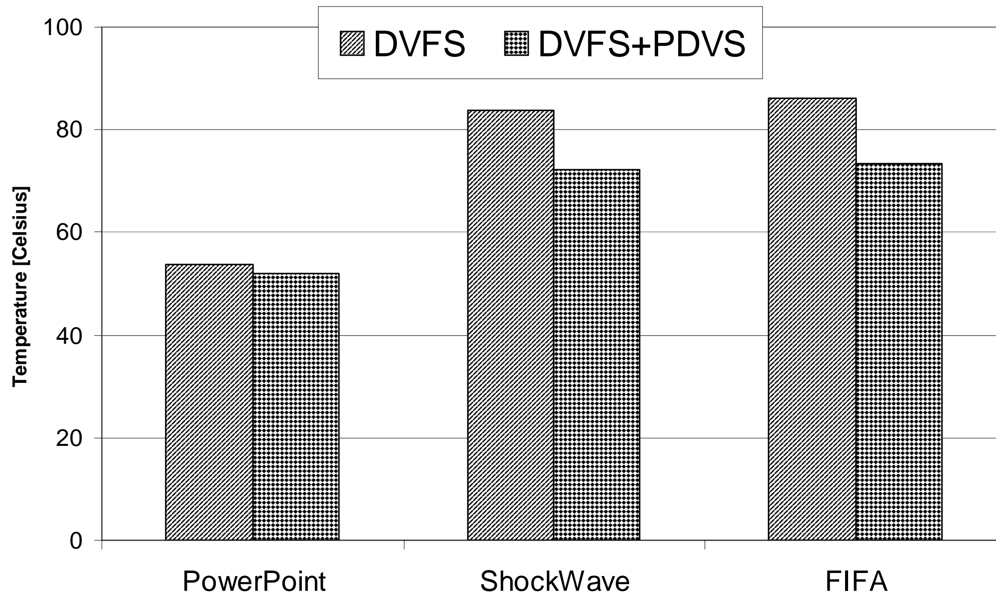
Figure 4.7. *Comparison of UDFS algorithms, UDFS+PDVS, and Windows XP DVFS (measured system power with display off). Chebyshev bound-based (1-p) values for difference of means from zero are also shown.*

On average, the power consumption of the overall system can be reduced by 49.9% for all three applications. This improvement is achieved by combining the UDFS2 scheme (22.1%) and PDVS scheme (27.8%).

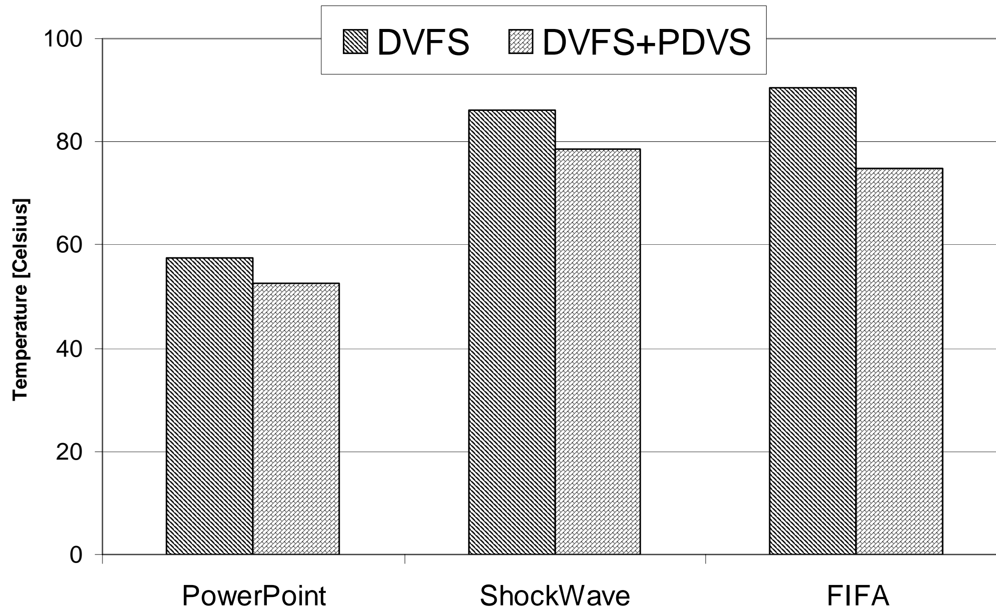
The results presented in the previous section, and in this section, cannot be directly compared because the previous section reports the simulated power consumption of the CPU and this section reports the measured power consumption of the laptop. However, some conclusions can be drawn from the data in both sections. For applications like PowerPoint, where the CPU consumes only a small fraction of the system power, the benefit on system power is low. On the other hand, for the applications that originally result in high CPU power consumption, the system power savings can be substantial due to the reduction in dynamic power as well as the operating temperatures and consequently leakage power.

4.3.5 Temperature

We used CPUCool [103] to measure CPU temperature in the system. Figure 4.8 shows the mean and peak temperatures of the system when using the different combinations of DVFS, PDVS, and UDFS schemes. The values reported for UDFS and UDFS+PDVS are the averages over 20 users.



(a) Mean Temperature



(b) Maximum Temperature

Figure 4.8. *Mean and peak temperature measurement.*

In all cases, the UDFS1 and UDFS2 schemes lower the temperature compared to the Windows native DVFS scheme due to the power reductions we have reported in the previous sections. The maximum UDFS temperature

reduction is seen in the case of the UDFS2 scheme used for the Shockwave application (7.0°C). On average, for all 3 applications, the UDFS1 and UDFS2 schemes reduce the mean temperature of the system by 1.8°C and 3.8°C, respectively. Similarly, PDVS reduces the mean system temperature by 8.7°C on average for the three applications. The best improvement is observed for the FIFA game, where temperature decreases by 12.6°C.

The combination of PDVS and UDFS is again synergistic, leading to even greater temperature reductions than PDVS or UDFS, alone. For the Shockwave application, UDFS2+PDVS reduces the mean temperature by 19.3°C. The average temperature reductions in all three applications by the UDFS1+PDVS and UDFS2+PDVS schemes are 12.7°C and 13.7°C, respectively. Our 13.2°C claim averages these two.

4.4. Discussion

We now discuss the degree of user interaction needed to make UDFS work, the CPU reliability and longevity benefits of our techniques, and the effects of multitasking.

Table 4-C. *Average number of user events.*

Algorithms	PowerPoint	3D animation	FIFA Game	
	4 min	4 min	4 min	4 min
UDFS1	0.35	11.85	5.10	3.42
UDFS2	0.60	14.25	6.50	3.82

4..4.1.2 User interaction

While PDVS can be employed without user interaction, UDFS requires occasional feedback from the user. Minimizing the required rate of feedback button presses while maintaining effective control is a central challenge. Our current UDFS algorithms perform reasonably well in this respect, but could be improved. Table 4-C presents the average number of annoyance button presses over a 4 minute period for both versions of UDFS algorithms in our 20 user study. Generally, UDFS2 requires more frequent button presses than UDFS1, because a single press only increments the frequency. The trade-off is that UDFS1 generally spends more time at the maximum frequency and thus is more power hungry. On average, a user pressed a button every 8 minutes for PowerPoint, every 18 seconds for the Shockwave animation, and every 50 seconds for the FIFA game. During the course of the study, for the 3D animation, there were some extreme cases in which the user kept pressing the button even when the processor was running at the highest frequency. This can be explained by the user's dissatisfaction with the original quality of the video or the maximum performance available from the CPU, over which we had no control. If we omit the three most extreme cases from both maximum and minimum number of annoyances, on average a user presses the annoyance button once every 30 seconds for the Shockwave application.

We also note that the system adapts to users quickly, leading to a reduced rate of button presses. In the Table 4-C, we show both the first and second 4 minute interval for the FIFA game. The number of presses in the second interval

is much smaller than the first. Our interpretation is that once a stable frequency has been determined by the UDFS scheme, it can remain at that frequency for a long time, without requiring further user interaction.

Table 4-D records the average number of voltage transitions for the six different schemes used in our study. A voltage transition is caused either due to a button press or a significant change in operating temperature. For the PowerPoint application, we observe a reduction in the number of transitions because the spikes observed for

Table 4-D. *Number of voltage transitions*

Applications	DVFS	DVFS+ PDVS	UDFS1	UDFS1+ PDVS	UDFS2	UDFS2+ PDVS
PowerPoint +Music	11.00	11.00	4.40	4.65	6.55	6.50
3D Animation	3.00	4.00	10.30	11.50	16.3	17.55
FIFA Game	6.00	6.00	18.06	18.05	28.85	29.30

DVFS do not occur for UDFS1 and UDFS2. On the other hand, the 3D animation and FIFA Game applications have more voltage transitions than observed with Windows native DVFS, because they aim to reduce power by adjusting throttle and, in effect, voltage. In contrast, conventional DVFS keeps the system at the highest frequency during the entire interval. The increase in the number of transitions for the PDVS schemes implemented on top of UDFS are

caused by the extra voltage transitions due to changing temperature at a given frequency level.

4..4.1.3 Reliability and longevity

In addition to its direct impact on power consumption, our techniques may ultimately improve the lifetime reliability of a system. Earlier research [111] showed that the effect of operating temperature on integrated circuit's mean time to failure (MTTF) is exponential. As we show in Section 4.3.5 , our schemes can reduce the operating temperature by 13.2°C on average, thereby potentially reducing the rate of failure due to temperature-dependant processes such as electromigration. Traditionally, the required supply voltage of a processor is reported at the maximum operating temperature of the system. Therefore, at temperatures below the maximum rated temperature, timing slack exists. As long as the current temperature is below the highest rated operating temperature, the operating voltage can be reduced below the rated operating voltage without reducing reliability below that of the same processor operating at the rated voltage and at the maximum temperature.

4..4.1.4 Multitasking

A natural question to ask is whether the extremely simple "press the button" user feedback mechanism we use in UDFS is sufficient for describing user preferences in a multitasking environment. To see the effect of UDFS in a multitasking environment, we conducted a small study (n=8) similar to that of Section 4.3. Instead of several consecutive tasks, the user was asked to watch a 3D

animation using Microsoft Internet Explorer while listening to MP3 music using Windows Media Player in compact mode with visualization.

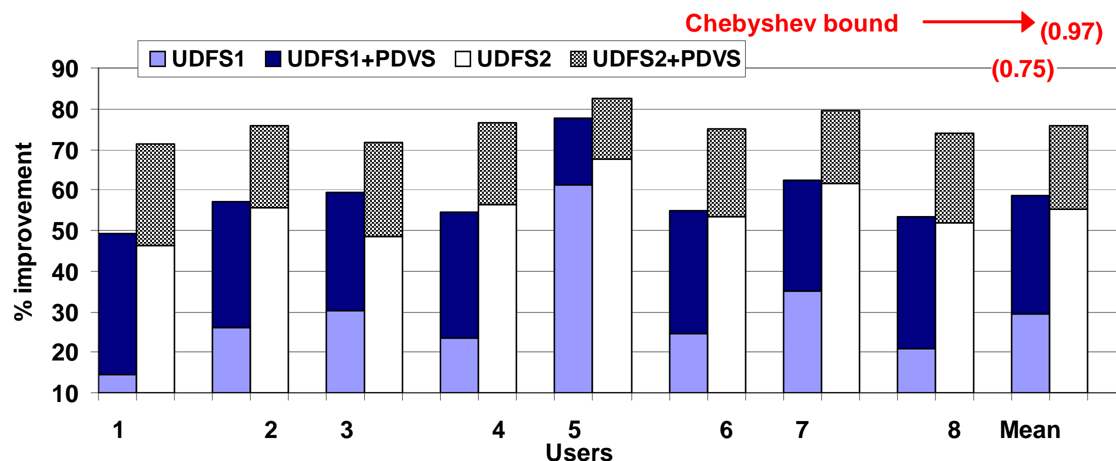


Figure 4.9. *Power improvement in the multitasking environment. Chebyshev bound-based (1-p) values for difference of means from zero are also shown.*

Figure 4.9 shows the measured system power improvements compared to Windows DVFS. On average, the power consumption of the overall system is reduced by 29.5% and 55.1% for UDFS1 and UDFS2, respectively. Adding PDVS improves the average power savings to 58.6% and 75.7% for UDFS1 and UDFS2, respectively. Although these results are preliminary, combined with the results from the combined PowerPoint+MP3 task described in Section 4.0.1, they suggest that the simple feedback mechanism is sufficient in a multitasking environment. It is clearly a better proxy of the user's satisfaction than the CPU utilization of the combined task pool.

4.5. Related Work

Dynamic voltage and frequency scaling (DVFS) is an effective technique for microprocessor energy and power control for most modern processors [112, 113].

Energy efficiency has been a major concern for mobile computers. Fei et al. [114] proposed an energy aware dynamic software management framework that improves battery utilization for mobile computers. However, this technique is only applicable to highly adaptive mobile applications. Researchers have proposed algorithms based on workload decomposition [115], but these tend to provide power improvements only for memory-bound applications. Wu et al. [116] presented a design framework of a run-time DVFS optimizer in a general dynamic compilation system. The Razor [117] architecture dynamically finds the minimal reliable voltage level. Dhar et al. [118] proposed adaptive voltage scaling that uses a closed-loop controller targeted towards standard-cell ASICs. These schemes are similar to the PDVS scheme. However, our approach is completely operating system controlled and does not require any architectural modifications and therefore incurs no hardware overhead. Intel Foxton technology [119] provides a mechanism for select Intel Itanium 2 processors to adjust core frequency during operation to boost application performance. However, unlike PDVS it does not perform any dynamic voltage setting.

Other DVFS algorithms use task information, such as measuring response times in interactive applications [120, 121] as a proxy for the user. Unlike Vertigo [122], we monitor the *user* instead of the application. Xu et al. proposed novel schemes [123] minimizing energy consumption in real-time embedded systems that execute variable workloads. However, they try to adapt to the variability of the workload rather than to the users. AutoDVS [124] is a dynamic voltage scaling

(DVS) system for hand-held devices. They used user activity as an indicator to detect computationally intensive CPU intervals and use that to drive DVS. In contrast, UDFS uses user activity to directly control the frequency of the system. Ranga et al. proposed energy-aware user interfaces [125] based on usage scenarios, but they concentrated on the display rather than the CPU. Gupta et al. [100] and Lin et al. [101] demonstrated a high variation in user tolerance for performance in the scheduling context, variation that we believe holds for power management as well. Anand et al. [126] discussed the concept of a control parameter that could be used by the user. However, they focus on the wireless networking domain, not the CPU. Second, they do not propose or evaluate a user interface or direct user feedback. To the best of our knowledge, the UDFS component of our work is the first to employ direct user feedback instead of a proxy for the user.

Dynamic thermal management is an important issue for modern microprocessors due to the high cost of cooling solutions. Previous work has discussed microarchitectural modeling and optimization based on temperature [127-130]. Liu and Svensson made a trade-off between speed and supply voltage [130]. Brooks and Martonosi [131] proposed dynamic thermal management for high-performance processors. For portable computers, Transmeta's Crusoe [132] and Intel's Pentium-M [113] are notable commercial products that uses innovative dynamic thermal management. To the best of our knowledge, the PDVS component of our work is the first to consider exploiting

process variation via per-CPU customization using profiling. In addition, it is the first scheme to consider temperature in voltage level decisions.

4.6. Conclusion

We have identified processor and user pessimism as key factors holding back effective power management for processors with support for DVFS. In response, we have developed and evaluated the following new, process- and user-adaptive DVFS techniques: process-driven voltage scaling (PDVS) and user-driven frequency scaling (UDFS). These techniques dramatically reduce CPU power consumption in comparison with existing DVFS techniques. Extensive user studies show that we can reduce power on average by over 50% for single task and over 75% for multitasking workloads compared to the Microsoft Windows XP DVFS scheme. Furthermore, CPU temperatures can be markedly decreased through the use of our techniques. PDVS can be readily used along with any existing frequency scaling approach. UDFS requires that user feedback be used to direct processor voltage and frequency control. PDVS and UDFS are synergistic. UDFS leads to lower average frequencies and PDVS allows great decreases in voltage at low frequencies.

CHAPTER 5

User-perceived performance evaluation

Existing architectures/systems typically aim at optimizing for user satisfaction by employing metrics based largely on instruction throughput (e.g., instructions-per-second). These metrics are used because they are easy to access, easy to compare across platforms, and are believed to reflect user demands for performance at a very low level. However, in this chapter, we will show that low-level information is not as good a proxy for user satisfaction with performance as is high-level information actually observed or perceived by the user. We focus on interactive applications and show that it is possible to infer information about user-perceived performance by measuring changes at the user interface. This provides better information about the performance level necessary to maintain user satisfaction and therefore can be used to achieve a reduction in power consumption, a reduction in heat generation, and an increase in lifetime reliability.

Processor frequency has a strong effect on power consumption and temperature, directly and also indirectly through the need for higher voltages at higher frequencies. Therefore, Dynamic Voltage and Frequency Scaling (DVFS) is one of the most commonly used power reduction techniques in modern processors.

DVFS varies the frequency and voltage of a microprocessor at runtime according to processing needs. Although there are many different versions of DVFS, at its core DVFS adapts power consumption and performance to the current workload of the CPU. Specifically, existing DVFS techniques in high-performance processors select an operating point (CPU frequency and voltage) based on the utilization of the processor and other information available to the Operating System (OS) kernel. This approach is pessimistic regarding user satisfaction and assumes that the maximum processor frequency is necessary for every process. A high level of CPU utilization or a burst of certain OS events leads directly to a high frequency (and hence high voltage), regardless of the user's satisfaction or expectation of performance. This can produce unnecessary increases in frequency, voltage, power, and temperature.

In response to this observation, discussed further in Section 5.1, we have developed a new power management technique that relies upon a more accurate proxy for user performance needs than CPU- or OS-level events, but is still inexpensive to measure. We propose to determine user satisfaction with processor performance not with information that is "close to metal" and hidden from the user, but rather with information that is "close to flesh" and apparent to the user. Interface devices are the logical locations for these measurements since they sit between computation and user perception. The display is particularly useful because it is the user's primary source of information regarding the performance of the computer.

We must note that a user-satisfaction aware optimization metric does not have to provide absolute values, but relative values are sufficient. In other words, we do not need an exact measure of user-perceived performance to make decisions. For example, consider an application, such as video playback, that only affects the screen. If there are two settings on the architecture that result in identical sequences and timing of frames on the screen, then we can safely conclude that these two states have the same performance. Using this idea in the context of DVFS, we can compare application performance (i.e., user-perceived performance based on changes in the display) to the performance measured using the same metric at the highest available frequency. Hence, we only need to make relative measurements to determine user satisfaction.

To bring this idea to life and evaluate it, we have developed a new power management framework called **PICSEL** (**P**erception**-I**nformed **C**PU performance **S**caling to **E**xtend battery **L**ife) that monitors the rate of change of pixel intensities in the display. An algorithm controlling the processor's operating frequency then makes decisions based upon these rates of change. The algorithm is tested with two configurations: conservative PICSEL (cPICSEL) and aggressive PICSEL (aPICSEL) (Section 5.2.2). We focus on the DVFS technique implemented by a commercial OS and show that runtime information on user-perceived performance can enhance the effectiveness of the power management scheme. We also show that this approach (i.e., considering user satisfaction while taking architectural decisions) can result in optimizations that are not possible

otherwise. This is a collaborative work with my colleague Jack Cosgrove who is a graduate student interested in correlating user satisfaction with computer display. Specifically, this work makes the following contributions:

- We show that traditional performance metrics do not necessarily represent user-perceived performance,
- We introduce new metrics that can successfully measure user-perceived performance, and
- We propose, implement, and evaluate PICSEL, a power management scheme that utilizes user-perceived performance.

The chapter is organized as follows. In Section 5.1, we describe the motivational results showing the difference between instruction-throughput and user-perceived performance. PICSEL is described in Section 5.2. Section 5.3 presents the results obtained from user studies. We compare our work with previous studies in Section 5.4. Section 5.5 summarizes our contributions.

5.1. User-Perceived Performance

The motivation for including user-perceived performance in any objective function is clear: any optimization (performance, power, reliability, security, etc.) ultimately aims to satisfy the user. However, the difficulty in optimizing directly for user-perceived performance is finding a metric that corresponds to it. For interactive applications, the events occurring on the input/output devices are good candidates for measuring what the user observes. However, input events are rare

compared to output events. Therefore, considering output to the user is preferable for estimating the performance experienced by the user. Of all the types of output supplied to the user, graphics are used in the highest proportion of applications. Therefore, utilizing properties of the display to measure user-perceived performance is a good alternative.

Given an application that only changes the display, it is safe to assume that the sequence of frames is an indication of the user-perceived performance. For example, if there are two architectural alternatives that result in identical frame timings and sequences, we can conclude that these architectures provide the same user-perceived performance. On the other hand, what happens if the sequences are different? One alternative would be to consider frame rate. For example, if the frame rate is decreased by 10%, then we may claim that the user-perceived performance is reduced by 10%. However, the correlation between frame rate and user satisfaction may be weak. In fact, Ghinea and Thomas [133] have done a perceptual study showing that neither frame rate nor color depth are significant predictors of user satisfaction, but the combination of these two entities strongly correlates to the user satisfaction. However, extracting the exact frame rate and color depth information would require changes in the application and/or OS. Hence, we decided to utilize a metric that is independent of the application and easily measurable: we measure the rate of display pixel change over time, which captures the combination of these two metrics.

It is important to note that measuring our metric, pixel rate change, has comparable complexity to measuring existing metrics such as CPU utilization that are commonplace in today's architectures. In addition, the link between user satisfaction and changes on the display is supported by prior work [133], as well as intuition.

Traditionally, the rate of instruction execution has been widely used as a measure of system performance. First, we perform a set of experiments measuring the instruction throughput and the rate of pixel change to understand the relation between these two metrics. In these experiments, we measure the number of instructions-per-second (IPS) on a 2.13 GHz Intel Pentium M-based laptop (please see Section 5.3 for further details on the experimental study environment) for three applications: a 3D Shockwave animation, a DVD quality video played, and a 3D video game. We also measured the changes in intensity in the red, green, and blue channels of some of the pixels being used to display these applications using the method described in Section 5.2.3, and averaged these changes together for each time instance to obtain the **Average Pixel Change (APC)**. The procedure to calculate APC is presented in Table 5-A. We repeat these measurements at all six available processor operating frequencies.

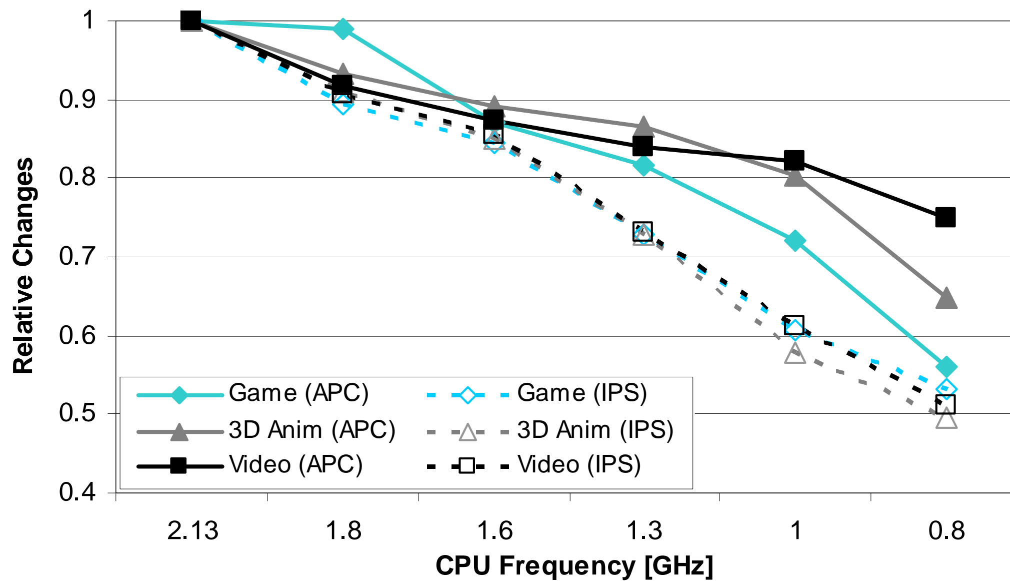


Figure 5.1. *IPS and APC curve*

Table 5-A. *User-Perceived Performance Metrics*

Metrics	Measurement Procedure
Average Pixel Change (APC)	- Capture the Pixel intensities of the RGB channels of all the pixels in a memory buffer - Calculate the relative changes for all the sampled pixels - The mean of relative changes is the APC
Rate of Average Pixel Change (APR)	$(APC_{Ti} - APC_{Ti-1})/(T_i - T_{i-1})$

Figure 5.1 illustrates the results of this experiment, with the solid lines representing the APC and dotted lines representing the IPS. As depicted in the figure, the IPS of a system is closely related to the operating frequency and is fairly uniform across the three applications. APC is also dependent on the operating frequency, but this dependence is influenced by the application more so than IPS. For the Shockwave application, the effect on APC due to frequency throttling is below 10% for the highest three frequencies. The Video application

shows similar properties. For this task, we could simply set the frequency statically to a lower value without causing noticeable change in the APC. For the game application, the highest two frequency states can sustain the APC value within the 10% threshold. However, the lower frequency states cause the APC value to drop suddenly. Most importantly, we see a significant difference between the reduction in IPS and APC. In other words, these results support our claim that the instruction throughput and user-perceived performance are not linearly related. We observe that the APC value of a system can quantize user perceived performance and can be used as a control parameter for a power management scheme that implements DVFS based on user-perceived performance.

The primary metric we use for user-perceived performance is APC normalized to the total number of pixels in the display. As shown in the Figure 5.1, we observe considerable variation in the APC values across different applications as well as different frequency states. On the other hand, it is also possible that the reduction in the frequency may result in discontinuities in the display. Previous researchers [134] have found that jitter and latency are the main sources of user discontent in networked multimedia applications. For example, consider an application that starts skipping frames when the computational power is reduced. In such a case, the APC may not be affected significantly: in a sequence of frames, even if some of the intermediate ones are skipped, the pixel difference between the first and the last does not change. To capture the occurrences of such discontinuities, we record the **Rate of Average Pixel**

Change (APR) normalized over the number of pixels. In other words, we calculate the difference between the APC values measured at each time instant. This roughly corresponds to the derivative of the APC. Figure 5.2 illustrates the APR trends observed in three applications used in this research project. When there are glitches during display, in general the APR value increases rapidly. This is true for applications where the glitch problem is observed at the lower frequencies, namely the Video and 3D Shockwave animation. For other applications (such as the game), we simply observe an overall slowdown and APR values drop in parallel to APC levels. This reduces game jitter at the price of slowing the entire game down. As a result, for this particular application we actually observe a reduction in APR value at lower frequencies as the game's average frame rate is reduced.

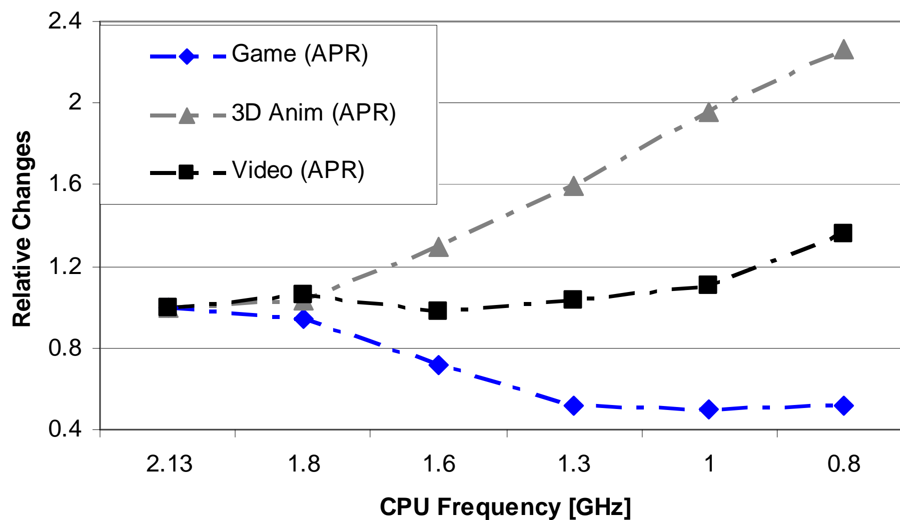


Figure 5.2. *APR curves for the three applications*

The APR reveals even more pronounced differential behavior. This behavior can permit a DVFS algorithm to differentiate between two applications with similar computational loads and to assign them to different operating frequencies, one potentially lower than would have otherwise been assigned by existing pessimistic DVFS schemes.

5.2. PICSEL Framework

User-perceived performance-based frequency scaling has two components. First, we have to measure the rate of change in the pixels displayed on the screen. This measurement tool is described in the next section. Then, we have to make a throttling decision based on these measurements. The algorithm making this decision is described in Section 5.2.2. In Section 5.2.3, we describe how PICSEL interacts with the system.

5.2.1 PICSEL Display Access

There are several methods for accessing the content of a computer display owing to the many steps involved in generating this content. Although more complex schemes are possible, the organization of a generic graphics pipeline in a contemporary computer is shown in Figure 5.3.

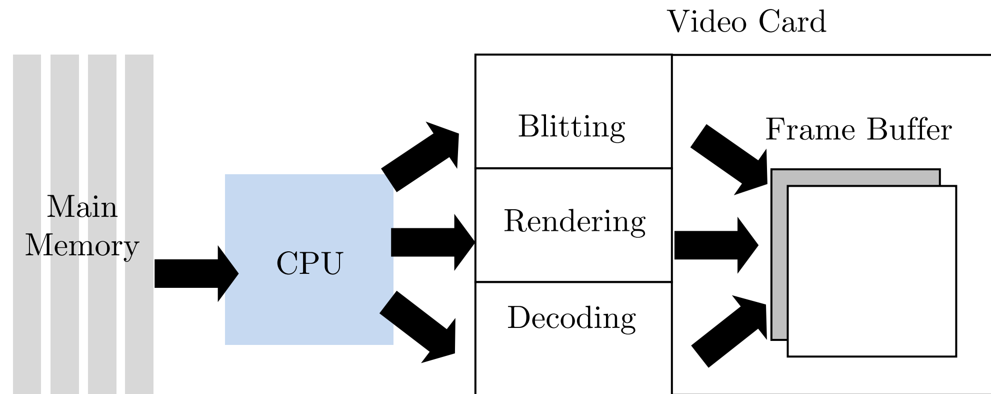


Figure 5.3. *Graphics pipeline in a modern PC*

Application content is read and produced by the CPU, which determines what action should be taken by the video card. The video card then performs operations on the data stream sent by the CPU. The most common operations are blitting, rendering, and decoding. Blitting is a method to erase and redraw sections of a bitmapped image more quickly than a raster scan. Rendering uses highly parallel floating-point processors to display three-dimensional primitives as two-dimensional projections. Video cards can also perform hardware-accelerated decoding of such compression algorithms as MPEG-2. Each of these different methods may use separate portions of video memory that are invisible to each other until composition on the frame buffer. The frame buffer consists of at least two video memory buffers each as large as the monitor screen upon which the separate video buffers are pieced together through a process called composition.

PICSEL gathers screen information using the Windows XP screenshot method, which is simple to implement and can blit any region of the screen to main memory. However, screen content may be missing from sections of the blitted

region if those sections were drawn elsewhere in video memory by a rendering or decoding operation. Such a sub-section of video memory stored outside the full screen video memory and later composited within a window is called a hardware overlay. Screen data from a hardware overlay can be obtained, however, by turning off the hardware overlay option in the application owning the overlay. The lack of hardware overlays does not degrade performance during testing. It should also be noted that the blitting method will be less available in the future as desktop environments move towards full rendering, e.g., Quartz on Macintosh, Aero on Windows, and Compiz Fusion on the X Server.

Ideally we would like to consider all the pixels present in the display while calculating the APC. Furthermore, the rate of APC calculation should be same as rate of frame change in the system. However, both of these constraints introduce heavy computational overhead on the system. Therefore, it is necessary to reduce the size of the captured screen area so that the capturing process does not occupy too much of the computer's resources (we decided to limit the overhead to less than 2% CPU utilization). The final captured area is 64 by 51 pixels, or a scaling down of each dimension of a 1280 by 1024 screen by a factor of twenty. This block contained 3276 pixels and was fixed at the center of the screen. Moreover the sampling frequency for calculating APC is set to 10 Hz. Increasing the sampling frequency further increases the computational overhead. As we will show in Section 5.3, for our target applications, these limitations do not prevent PICSEL from capturing the user-perceived performance. Nevertheless, it is possible that

applications will not use our focus area; hence it may be desirable to overcome these limitations for other application domains. There are two design alternatives to solve this problem. First, the PICSEL algorithm can be implemented in hardware (either the CPU or the graphics card). The simplicity of the algorithm ensures relatively easy implementation in actual hardware at low overhead. Second, PICSEL could be executed on the graphics hardware. Although such implementations would be desirable, our goal in this work is to provide a proof-of-concept, which is achieved with the current implementation of PICSEL.

After a section of the screen has been captured, it is stored to a memory buffer. This buffer is compared to another buffer containing the previous screen capture, and the magnitudes of the intensity differences for the red, green, and blue channels are calculated. Only two buffers are necessary, with each buffer toggling between old and new screen captures. All of the magnitude differences are summed together to obtain a single statistic describing the first time derivative of pixel intensity over the sampling period (APC).

It is important to understand that this method does not capture each frame. However, since we also measure the APR, the difference between the frame rate and the sampling rate does not prevent us from capturing any slowdown because the jitter of pixel intensities will be captured with the APR metric. It is this second time derivative that permits a sampling frequency below the frame rate of the screen.

5.2.2 PICSEL Algorithm

PICSEL decides on the frequency level by using three state variables: f, the current CPU frequency; μ_{APC}, APC in the last time interval; and μ_{APR}, APR in the last time interval. Pixel data are measured at fixed sampling frequency and stored to a file by a background process. Adaptation is controlled by three constant parameters: ρ, the APC change threshold; γ, the APR change threshold; and α, the threshold difficulty level corresponding to each frequency state. PICSEL can either be in the initialization or the control state. The idea in the initialization stage is to capture information about the APC and APR values observed at the highest frequency. These values will be compared against during the control stage to make throttling decisions. Therefore, during initialization, the CPU frequency is set at the highest value f_{max} for a time interval T_{init}. The APC and APR values of the system over the time interval T_{init} are obtained from the background process and initialized as APC_{global} and APR_{global}. PICSEL then enters the control state where at the end of each time interval T_i, the APC and APR of the system over the last interval is obtained from the background process. PICSEL then makes a decision as follows:

```
If     μAPC  <  ( 1-ρ*(1-α) )*APCglobal or
       μAPR  <  ( 1-γ*(1-α) )*APRglobal
{
   Reduce f by one level;
   Reset α of the last level to 0;
}
Else {
   Increase f by one level;
   Increment α;
}
```

The main idea in this code is to compare the last observed APC and APR against the "global" APC and APR (i.e., APC and APR captured when the processor is executing at the highest frequency). Then, based on the threshold factors defined by and we conclude that the user-perceived performance is unchanged and try to reduce the frequency and subsequently power consumption. Otherwise, out of bound values of $_{APC}$ and $_{APR}$ suggest that user-perceived performance has suffered on the last interval due to low CPU frequency and it is increased accordingly to improve the user-perceived performance.

The goal of the factor is to eliminate the possible ping-pong effect between two frequency states. If the processor has been at a state several times after which PICSEL had to increase the frequency, makes it harder to go down to that frequency level. Following every third (n=3) update to , PICSEL reenters the initialization state. This feature of the algorithm ensures that PICSEL can adjust to a set of operating conditions very different from those present at initialization but at a rate that is maximally bounded by n and Ti. The constant parameters (T_i = 7 seconds, T_{init} = 10 seconds) were set based on the experience of the authors using the system. α is initialized to zero for each of the frequency level and is incremented by 0.1 for each frequency boost. We used two variations of the PICSEL algorithm by fixing the and which correspond to **conservative PICSEL (cPICSEL)** and **aggressive PICSEL (aPICSEL)**, respectively.

Ideally, we would like to empirically evaluate the sensitivity of PICSEL performance to these parameters. However, it is important to note that any such

study would require having real users in the loop, and thus would be quite slow. Testing three values of five parameters on 20 users would require 243 days (based on 20 users/day and 25 minutes/user). For this reason, we decided to choose the parameters based on qualitative evaluation by the authors and then “close the loop” by evaluating the whole system with the choices.

5.2.3 Implementation/Integration of PICSEL

Currently, we have not integrated PICSEL with the OS, rather for our user studies, we manually give control to PICSEL. Once PICSEL is active, it executes client software that runs as a Windows toolbar task as well as an API that controls CPU frequency based on user perceived performance. In the client, we log the APC and APR at the background. The API uses these values to control CPU frequency. It is this implementation that we evaluate in the next section.

In its current implementation, PICSEL has some limitations, which will be handled once it is integrated with the OS. Particularly, PICSEL should be activated only if the system is executing an interactive application. Hence, we first have to deal with detecting interactive applications, which will be handled through constant (but infrequent) monitoring of the device. For example, a daemon can monitor the APC/APR values every 10 seconds. Then, if this rate is above a threshold, we conclude that the current foreground application is an interactive one and activate the PICSEL frequency control. If the APC/APR value drops below a threshold, PICSEL will conclude that the interactive application is completed and give the control back to the Windows DVFS. In scenarios when the

display is static, PICSEL will detect that the rate of change in the display is below the threshold and give the control back to the Windows DVFS. On the other hand, if the machine runs a screen saver that causes significant changes on the screen, PICSEL will control the frequency. In such a case, the frequency will be reduced if there is no background job (and hence the reduction in the frequency does not cause a significant change on APC/APR), but will keep it high if a background job keeps the CPU busy (hence a change in the frequency will cause a significant change in APC/APR).

We must note that running background jobs does not cause any problem for PICSEL. In fact, one of our applications targeted in the next section includes a non-interactive background job to prove that our concept is applicable in such cases. If there is a CPU-intensive background job, a reduction in the frequency causes a significant reduction in the APC (even if the interactive application itself is not compute intensive). Therefore, PICSEL will keep the frequency high. If, on the other hand, the background job is not CPU-intensive, the frequency can be safely reduced, which is exactly the action taken by PICSEL.

5.3. Evaluation

We now evaluate cPICSEL and aPICSEL schemes. We compare against the native Windows XP DVFS scheme, displaying reductions in power and temperature. In Section 5.3.7, we also present the results summarizing the user satisfaction.

Our evaluations are based on user studies, as described in Section 5.3.1. We trace the user's activity on the system during the use of the applications and monitor the selections Windows DVFS, cPICSEL, or aPICSEL makes in response. For studies involving PICSEL, the cPICSEL and aPICSEL algorithms are used online to control the clock frequency in response to APC and APR values. In the rest of this section, we first describe a user study of PICSEL that provides both independent results and traces for later use. Next, we examine dynamic CPU power consumption, system power measurements (for a system driven from the user traces), and temperature measurements.

PICSEL effectively employs user-perceived performance via APC and APR values and customizes processor frequency to the individual user. This typically leads to significant power savings compared to existing dynamic frequency schemes that rely only on CPU utilization as feedback. The frame buffer readings and the corresponding calculations for measuring user-perceived performance are infrequent, and generally results less than 2% computational overhead. We must note that PICSEL performs the APC and APR readings during user studies, hence all the results presented for the PICSEL (including power and user satisfaction) include this overhead. We would also like to point that a more efficient implementation or hardware support from the graphical interface would minimize this overhead and would increase the benefits observed from PICSEL even further.

5.3.1 Experimental Setup

Our experiments were done using an IBM Thinkpad T43p with a 2.13 GHz Pentium M-770 CPU and 1 GB memory running Microsoft Windows XP Professional SP2. The Pentium M uses the second generation of Intel's SpeedStep technology, in which six CPU frequency-voltage operating points are available.

In all the studies, we make use of three application tasks, some of which are CPU intensive and some of which frequently block while waiting for user input:

- Watching a 3D Shockwave animation using the Microsoft Internet Explorer web browser. The animation was stored locally. Shockwave options were configured so that rendering was done entirely in software on the CPU.
- Playing the FIFA 2005 Soccer game. FIFA 2005 is a popular sports game. The game was stored locally. There were no constraints on user gameplay.
- Watching an HD quality movie trailer in Windows Media Player (WMP) while decoding another MPEG movie clip in the background. Both clips were stored locally and decoding was done in software on the CPU.

We conducted a study with twenty users to evaluate the PICSEL schemes. We developed a user pool by advertising our studies within a private university. Some participating users were graduate students and some others were less experienced with computer use. The studies were double-blind and randomized (i.e., the order of schemes during the tests were randomized to eliminate any possible effect of "first-time" execution impact). The studies included intervention

by proctors between trials. Each user evaluation lasted about thirty minutes, and consisted of the user doing the following:

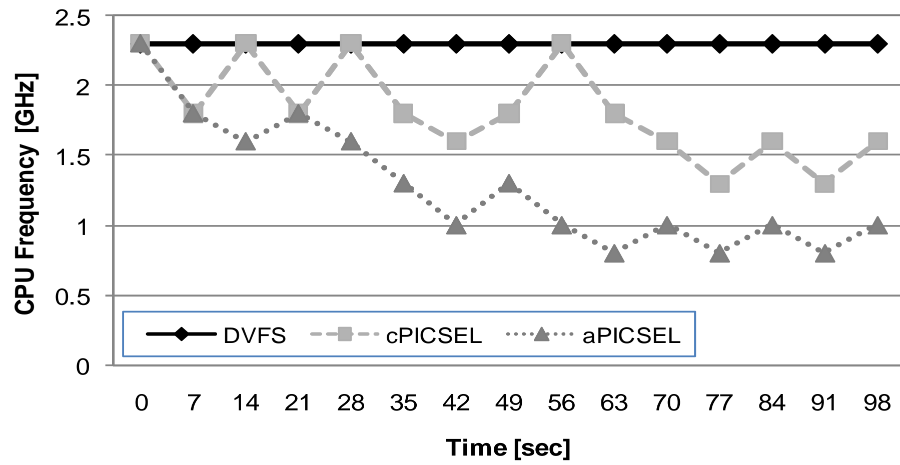
- Filling out a questionnaire that asked the user to rate his or her level of experience in the use of PCs, Windows XP, DVD video, 3D animation, and FIFA 2005
- Listening to an explanation of how to play FIFA 2005 and how to rate his or her satisfaction with each application instance
- Watching the 3D Shockwave animation using cPICSEL, aPICSEL, and Windows DVFS (2 minutes each)
- Playing FIFA 2005 using cPICSEL, aPICSEL, and Windows DVFS (3.5 minutes each)
- Watching the movie trailer using cPICSEL, aPICSEL, and Windows DVFS (2 minutes each).

After each application, the users were instructed to assign one of five levels of satisfaction to their experiences with the system performance for each instance of an application. In other words, the users were not asked to rank the instances against each other.

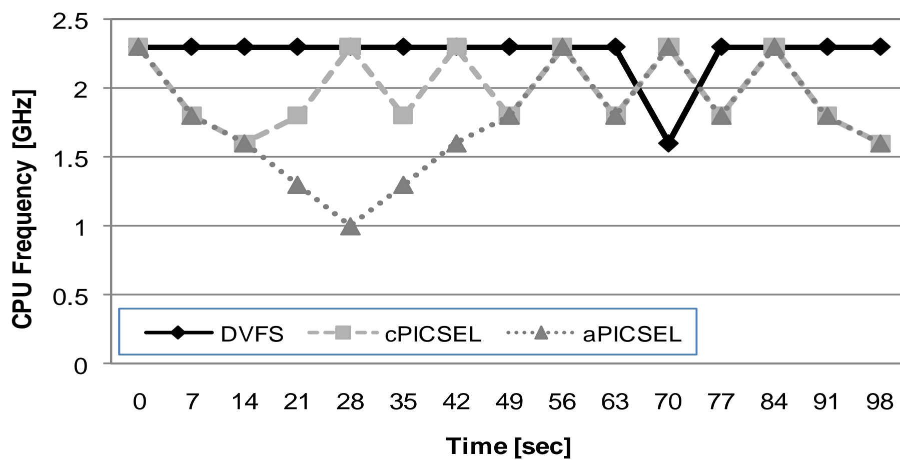
5.3.2 Frequency Results

Figure 5.4 illustrates the performance of the two algorithms for three applications in our study. Each graph shows, as a function of time, the CPU

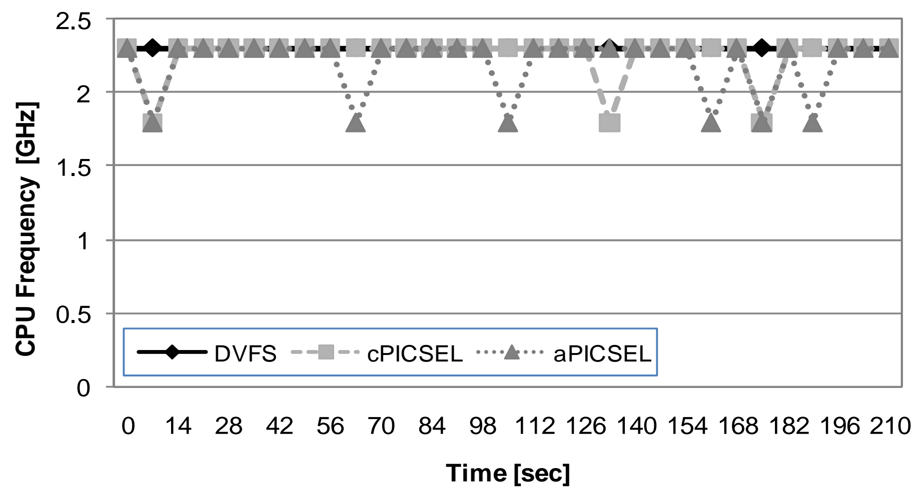
frequency for a randomly selected user (other users show the same trends although the exact values may be different). Notice that in all the applications, both versions of PICSEL were able to throttle down the processor as compared to the Windows DVFS scheme. The amount of frequency reduction varies from application to application. PICSEL is most effective for the 3D animation application. As illustrated in Figure 5.1and Figure 5.2, the 3D animation has least variation in APC and APR values at lower frequencies. As a result the PICSEL algorithm could reduce the CPU frequency to the lower states without affecting the user-perceived performance. The video application follows a similar trend. For the game, we observe little throttling. This is also expected as the APC values in Figure 5.1 degrade very quickly for the game. However, PICSEL algorithm can throttle down the frequency to lower frequency states in few cases. Overall, these results show that PICSEL can successfully adjust the throttling according to the user-perceived performance. Particularly, for a highly compute-intensive application (such as the game), the reduction in the frequency remains minimal. For other applications, the frequency can be reduced without affecting the user-perceived performance. In Section 5.3.7, we also analyze the user satisfaction with the default Windows DVFS (which almost always uses the highest frequency) and PICSEL algorithms and show that the user happiness is not adversely affected for any of our target applications.



(a) 3D Shockwave animation

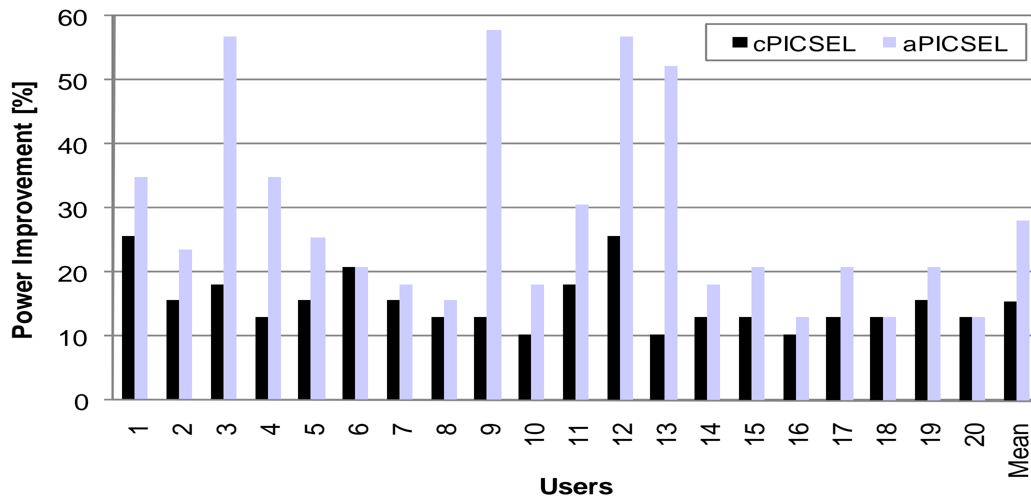


(b) Video

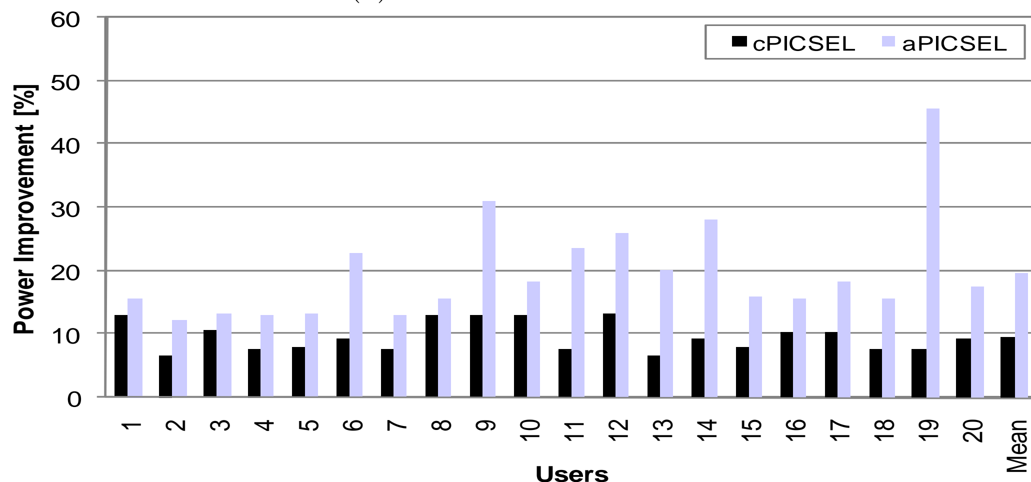


(c) FIFA game

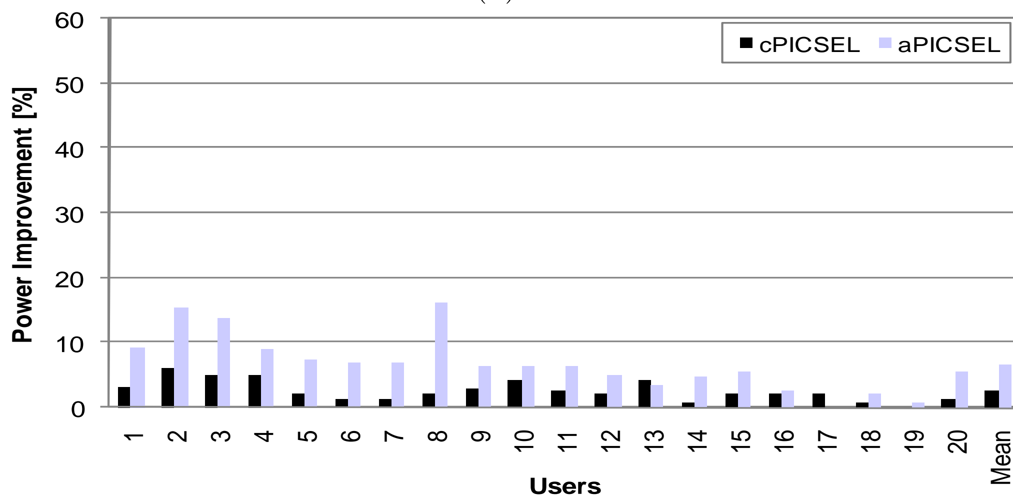
Figure 5.4. *Frequency state diagram*



(a) 3D Shockwave animation



(b) Video



(c) FIFA game

Figure 5.5. *The CPU dynamic power reduction with cPICSEL and aPICSEL over Windows DVFS*

5.3.3 Power Measurements

To analyze the effect of cPICSEL and aPICSEL on the power consumption of the system, we logged the frequency over time during the user studies described in the previous section. We then combine this frequency information with the offline profile and techniques described in Section 5.3.1 to derive CPU power savings for cPICSEL, aPICSEL, and the default Windows XP DVFS strategy. We have also measured the power consumption of the overall system, as described in Section 5.3.5.

5.3.4 CPU Dynamic Power Reduction

The dynamic power consumption of a processor is directly related to its frequency and supply voltage and can be expressed using the formula $P = V^2CF$, which states that power is equal to the product of voltage squared, capacitance, and frequency. By using the frequency traces and the nominal voltage levels on our target processor [113], we calculated the relative dynamic power consumption. Figure 5.5 presents the CPU dynamic power reduction achieved by the PICSEL algorithms (cPICSEL and aPICSEL) for individual users. The rightmost bars correspond to the savings averaged across users.

For the 3D Shockwave animation, we see mixed responses from the users, although on average we reduce power by 21.8%. On average, cPICSEL and aPICSEL independently reduce the power consumption by 15.3% and 28.2%, respectively. aPICSEL performs better as it allows a larger threshold for APC values over each interval. The results show a considerable variation among

different users. This can be explained by the fact that the control agent for APC calculation considers a sampling window of roughly 64x51 pixels at the center of the display window. The relative position of the shockwave player while the user watches the 3D animation plays a role in the calculation of APC and APR. It subsequently affects the decision taken by the PICSEL algorithm.

For Video, cPICSEL and aPICSEL reduce power consumption by an average of 9.6% and 19.7%, respectively. This suggests that the Video application is more sensitive to the frequency throttling. User 19 is the only exception where aPICSEL results a power savings of 45.8%. There is also considerable variation among users for the FIFA game. Using conventional DVFS, the system always runs at the highest frequency. The PICSEL schemes try to throttle down the frequency over time. They therefore reduce the power consumption while achieving reduction in power, on average 2.6% and 6.7% for cPICSEL and aPICSEL, respectively. Note that PICSEL does not reduce the frequency for all the users. For example, cPICSEL does not reduce the frequency for user 19. Similarly, aPICSEL does not reduce the frequency for user 17. This is expected as for the game application for which the slope for the APC curve (Figure 5.1) is most steep and for these users the change in APC and APR never satisfied the threshold condition for frequency reduction.

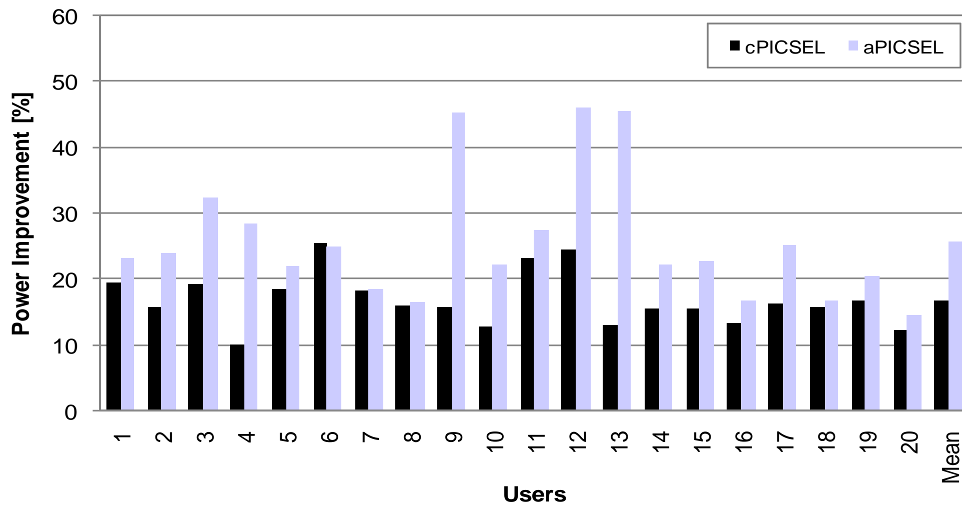
For all three applications, we see a large variation among users, but in all cases cPICSEL and aPICSEL lead to power savings over Windows DVFS. On average, aPICSEL reduces the dynamic power consumption by 18.2% for all three

applications. The cPICSEL scheme results a 9.1% power reduction aggregated over three applications and 20 users.

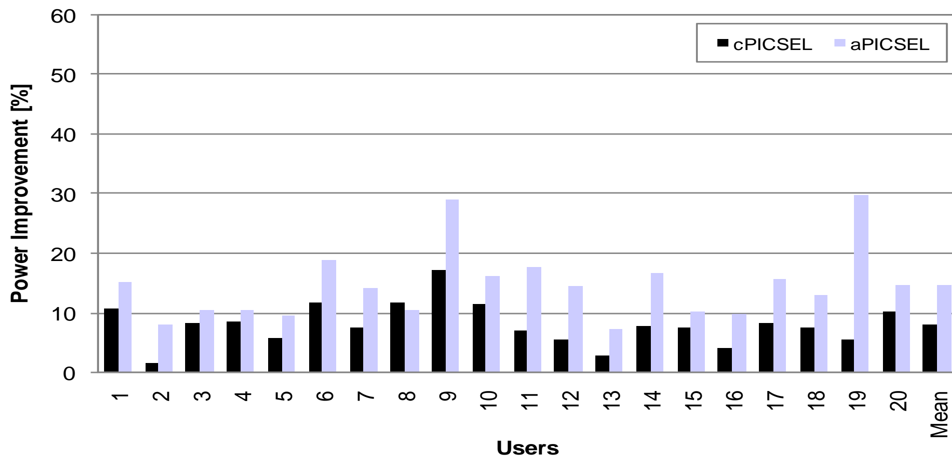
Figure 5.6. *System power measurement setup*

5.3.5 System power measurement

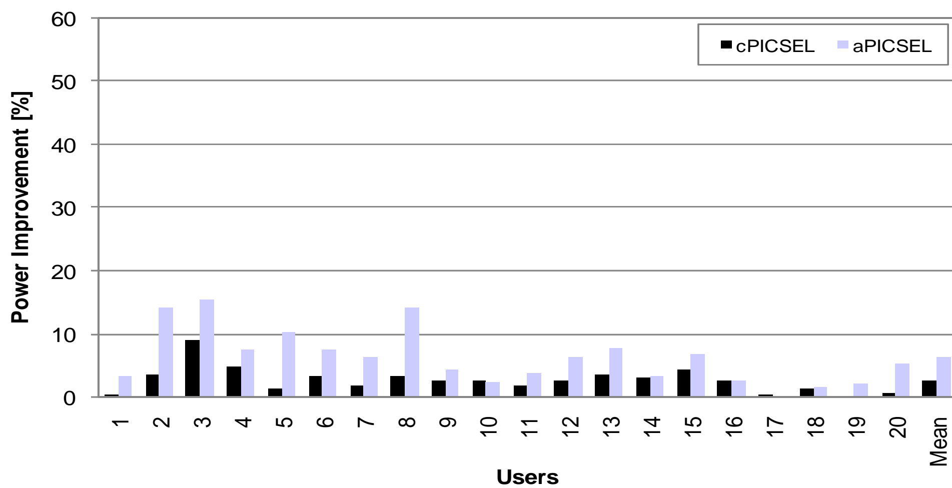
To further measure the impact of our techniques, we replay the traces from the user studies described in Section 5.3.1 on the laptop platform. The laptop is connected to a National Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Windows (and the target applications), which permits us to measure the power consumption of the entire laptop (including other power consuming components such as memory, screen, hard disk, etc.). The sampling rate is set to 10 Hz. Figure 5.6 illustrates the experimental setup used to measure the system power.



(a) 3D Shockwave animation



(b) Video



(c) FIFA game

Figure 5.7. *The system power reduction with cPICSEL and aPICSEL over Windows DVFS*

Figure 5.7 presents the system-level power savings over the default Windows DVFS for cPICSEL and aPICSEL schemes. In general, we see that the power savings of the system exhibits the same trends observed for dynamic power savings. For 3D Shockwave animation, cPICSEL and aPICSEL reduce power consumption by 16.8% and 25.7% on average, respectively. cPICSEL and aPICSEL reduce the power consumption by 8.0% and 14.5%, respectively for Video. For both the 3D animation and the Video, we see large variation among users. The FIFA game shows less variation among users. The high CPU overhead of this application restricts the PICSEL algorithms to throttle down the frequency. On average, we save 2.6% and 6.2% of the power consumption for cPICSEL and aPICSEL, respectively.

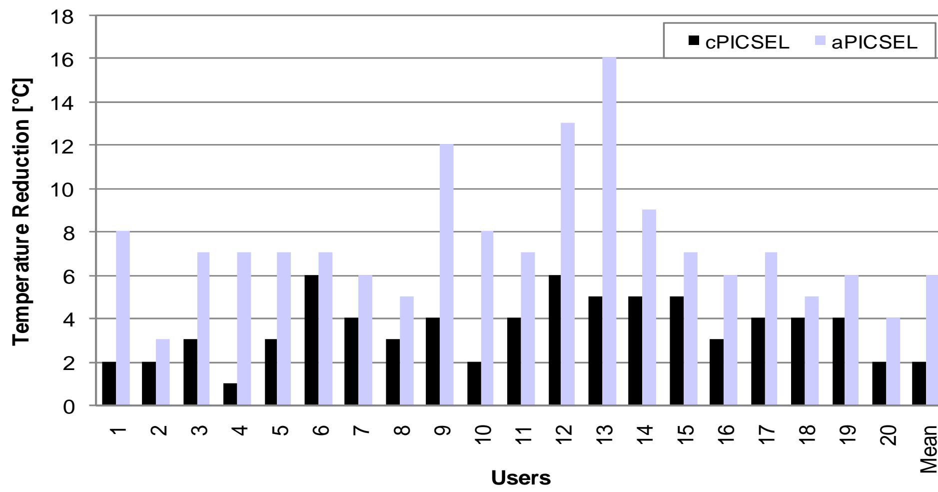
On average, the power consumption of the overall system can be reduced by 12.1% for all three applications. This improvement is achieved by the aPICSEL scheme. The cPICSEL scheme reduces the system power consumption by 7.1%, aggregated over 20 users and three applications. We must note that the dynamic CPU power savings presented in the previous section and the system power savings presented in this section cannot be directly compared because the previous section reports the dynamic power consumption of the CPU. This section, on the other hand, reports the measured power consumption of the laptop (which includes leakage power of the CPU as well as all the power consumption of other components in the laptop including memory, screen, hard disk, etc.). However, some conclusions can be drawn from the data in both sections. Applications that

originally result in high CPU power consumption tends to also observe high system power savings. Clearly, part of system power reduction comes from the decrease in the dynamic power consumption, but also the leakage is reduced when dynamic power consumption decreases (and hence temperature drops down).

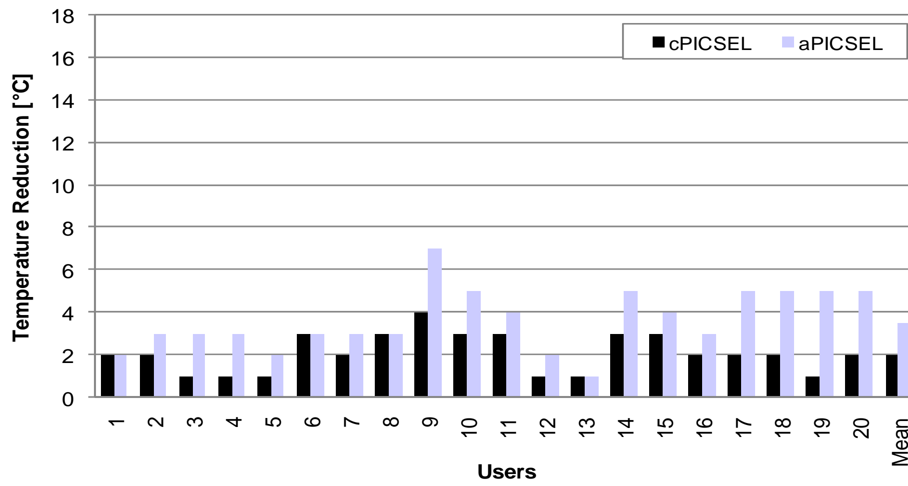
5.3.6 Changes in Peak Temperature

We used CPUCool [135] to measure CPU temperature in the system. Figure 5.8 shows the reductions in peak temperatures of the system when using the cPICSEL and aPICSEL schemes.

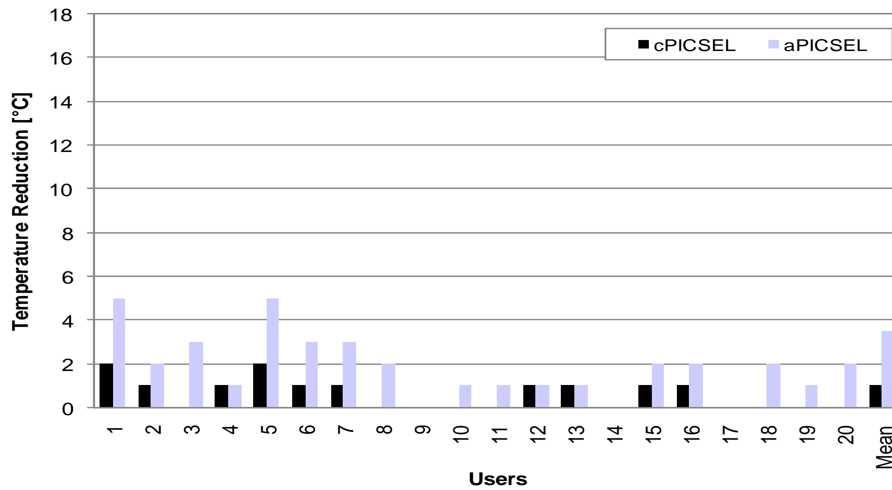
In all cases, the cPICSEL and aPICSEL schemes lower the temperature compared to the Windows native DVFS scheme due to the power reductions we have reported in the previous sections. The maximum temperature reduction is seen in the case of the aPICSEL scheme used for the Shockwave application (16°C). On average, for all three applications, the cPICSEL and aPICSEL schemes reduce the peak temperature of the system by 1.7 °C and 4.3°C, respectively, aggregated over all 20 users.



(a) 3D Shockwave animation

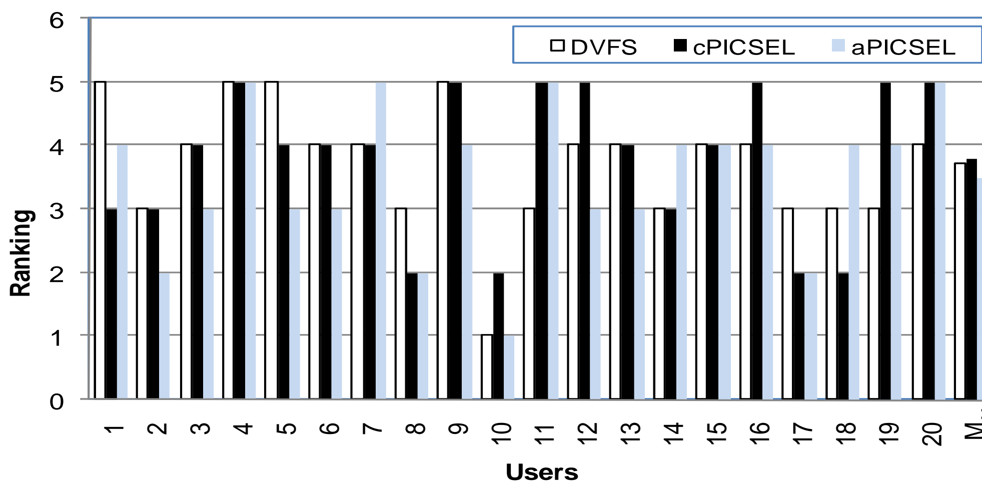


(b) Video

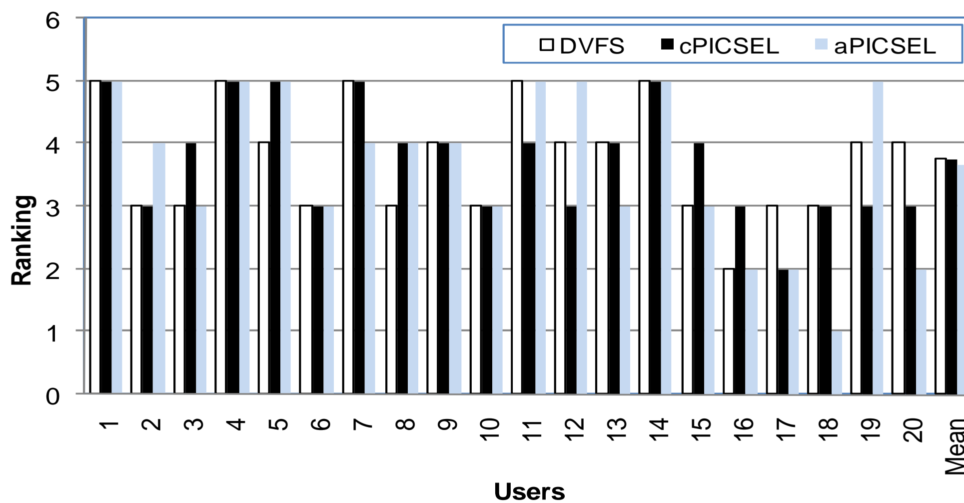


(d) FIFA game

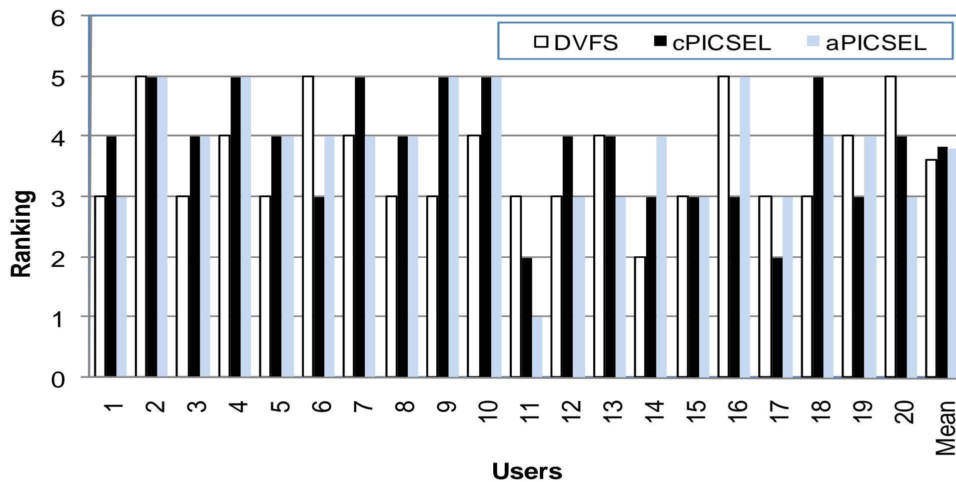
Figure 5.8. *Peak temperature reduction.*



(a) 3D Shockwave animation



(b) Video



(c) FIFA game

Figure 5.9. *User ranking distribution.*

5.3.7 User Satisfaction

We now discuss the satisfaction levels with the Windows DVFS and PICSEL algorithms for three applications as reported by individual users. During the user study each participant was asked to give a satisfaction level from 1 to 5 (5 being the most satisfactory performance) for each application. Figure 5.9 illustrates the ranks awarded by every user. cPICSEL algorithm outperforms Windows DVFS for all three applications when aggregated over 20 users. The t-test analysis of the results reveals that the difference is not due to chance with 90% confidence (i.e., cPICSEL statistically makes users happier). In the following, we will describe the reasons for these results. On the other hand, aPICSEL and Windows DVFS provide the same satisfaction (a t-test analysis identifies the two means to be identical with over 99% confidence). On average, aPICSEL scheme is rated highest for the game application (3.8) where it results least amount of power reduction. On the other hand, for the Shockwave application, maximum power reduction for the aPICSEL scheme came at a cost of least average rank (3.5).

We noticed that cPICSEL scheme was ranked higher on average when compared to Windows DVFS although Windows DVFS runs the system at the highest frequency. User dissatisfaction caused by thermal emergencies is the main reason for this outcome. We ran an experiment in which FIFA 2005 was played under Windows DVFS until the user observed several distinct slowdown events. The results of this experiment are shown in Figure 5.10. This figure shows processor temperature and frequency when FIFA 2005 is played until it triggers a

thermal emergency (about 1 minutes into the execution). At that point, the frequency is reduced to the lowest frequency. This causes a perceivable slowdown in game play. Once the emergency is over, the maximum allowed frequency is temporarily set to the second highest frequency. If the frequency line is followed, it is clear that this period lasts for about 30 seconds, after which the maximum allowed frequency is again set to the highest available frequency on the processor. This causes the temperature to rise again quickly and cause the consecutive emergencies.

We have analyzed the traces from the user studies and found the time spent in the second-highest operating frequency. The amount of time spent (for both Windows DVFS and PICSEL) provides additional evidence for thermal emergencies, since the only time DVFS will throttle the frequency below what CPU utilization would prescribe is in the case of the temperature crossing a thermal trip point [136].

Since cPICSEL and aPICSEL reduce the occurrence of thermal emergencies, PICSEL was able to provide a better user satisfaction, as emergency related frequency reductions are minimized. As a result, for applications with high computational overload, the PICSEL scheme may deliver better user-perceived performance by reducing the probability of thermal throttling in the CPU. The satisfaction results also support this claim: aPICSEL provides the highest satisfaction for the game on average, because for this highly computationally

intensive application, aPICSEL allows the highest reduction in temperature (and related emergencies).

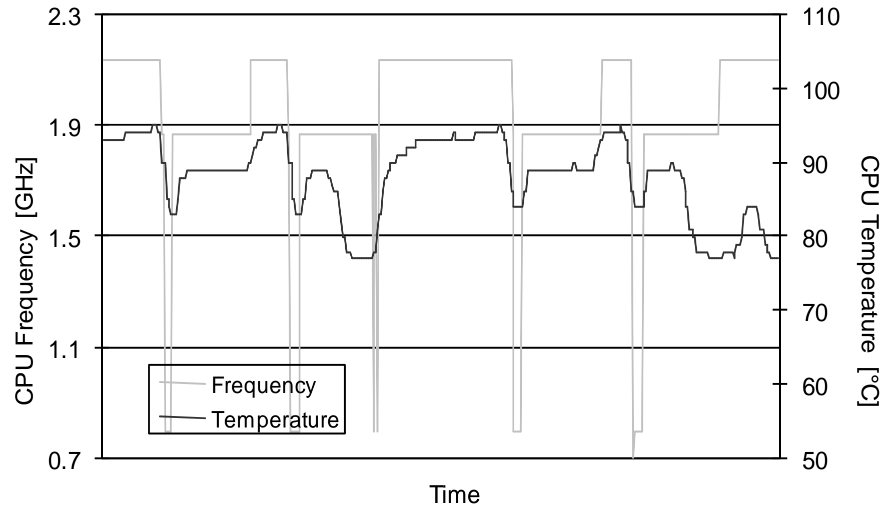


Figure 5.10. *Thermal emergency under Windows DVFS*

5.4. Related Work

Dynamic voltage and frequency scaling (DVFS) is an effective technique for microprocessor energy and power control for most modern processors [113], [112]. Energy efficiency has been a major concern for mobile computers. Fei et al. [114] proposed an energy aware dynamic software management framework that improves battery utilization for mobile computers. However, this technique is only applicable to highly adaptive mobile applications. Researchers have proposed algorithms based on workload decomposition [115], but these tend to provide power improvements only for memory-bound applications. Wu et al. [116] presented a design framework of a run-time DVFS optimizer in a general dynamic compilation system. The Razor [117] architecture dynamically finds the minimal reliable voltage level. Dhar et al. [118] proposed adaptive voltage scaling that uses

a closed-loop controller targeted towards standard-cell ASICs. Intel Foxton technology [119] provides a mechanism for select Intel Itanium 2 processors to adjust core frequency during operation to boost application performance. However, unlike PICSEL it does not perform any dynamic voltage setting. To the best of our knowledge, none of the previous DVFS techniques consider the user-perceived performance.

Other DVFS algorithms use task information, such as measuring response times in interactive applications [120] and [121] as a proxy for the user. Vertigo [122] monitors application messages and can be used to perform the optimizations implemented in our study (although to the best of our knowledge this has not been studied). However, compared to Vertigo, our approach provides a much easier metric/framework to use. Xu et al. proposed novel schemes [123] minimizing energy consumption in real-time embedded systems that execute variable workloads. However, they try to adapt to the variability of the workload rather than to the users. Gupta et al. [100] and Lin et al. [101] demonstrated a high variation in user tolerance for performance in the scheduling context, variation that we believe holds for power management as well. In addition, Mallik et al. [137] showed that it is possible to utilize direct user feedback to control a power management scheme, i.e., allow the user to control the performance of the processor directly. However, such a scheme has the potential to annoy the user while gathering feedback, whereas our scheme relies on inferring the user-perceived performance and hence can be applied transparently in the system. Anand et

al. [126] discussed the concept of a control parameter that could be used by the user. However, they focus on the wireless networking domain, not the CPU. Second, they do not propose or evaluate a user interface.

Previous work [125] has explored using OS-level knowledge about screen content to reduce the power consumption of the screen itself, however no work has been done using knowledge of screen content to control the voltage and frequency of a processor. Another work [124] has looked at OS-level knowledge of user-generated events to control a DVFS scheme but has not used knowledge of screen content. Our work combines these two approaches and uses detailed screen information to control the CPU's voltage and frequency levels.

In a study of user perception of both audio and video quality, it is found that the loss of several consecutive video frames would decrease user satisfaction up to a certain level, while an accumulation of video losses over the course of a video would steadily decrease user satisfaction [138]. User dissatisfaction at variations in the frame rate lay in between. Frame rate also has a significant effect on user satisfaction, with satisfaction increasing logarithmically with the number of frames displayed per second [139]. Finally, Gulliver and Ghinea found that both video delay and jitter cause a significant reduction in users' perception of the quality of a video [134]. However, none of these results were utilized to control processor resources.

5.5. Conclusion

Any architectural optimization (performance, power, reliability, security, etc.) ultimately aims to satisfy the user. The success of such an optimization relies upon the accuracy of its performance metrics as proxies for user satisfaction. In this work, we argue that rather than using metrics "close to metal" (such as instruction throughput or CPU utilization), architectures should optimize for metrics that are "close to flesh". To evaluate such an approach, we have developed a new power management technique: **PICSEL** (**P**erception**-I**nformed **C**PU performance **S**caling to **E**xtend battery **L**ife). This technique reduces CPU power consumption in comparison with existing DVFS techniques. Extensive user studies show that we can reduce power consumption of our target laptop on average by 7.1% for a conservative approach (cPICSEL) and 12.0% for the aggressive version (aPICSEL) compared to the Windows XP DVFS scheme. Furthermore, CPU temperatures can be markedly decreased through the use of our techniques. User studies also revealed that the difference in overall user satisfaction between the more aggressive version of PICSEL and Windows DVFS were statistically insignificant, whereas the conservative version of PICSEL actually improved the users' overall satisfaction when compared to Windows DVFS.

CHAPTER 6

Contributions and Conclusions

We have presented the framework for holistic architecture that optimizes system performance by utilizing characteristic of the applications, users and materials. We have shown that holistic architecture is capable of producing system architectures that is not achievable using traditional microarchitectural optimization.

The *main contribution* of the holistic architecture is *inclusion of additional abstraction layers into the whole spectrum of computer architecture*. Traditionally computer architecture tries to optimize system performance using resources that are 'within the box'. However, the user is at the top of the pyramid of all the computing system structure. Although he is not a part of the box, his satisfaction is the ultimate objective of every architectural optimization. To the best of our knowledge, our work [23, 140, 141] was the *first* one to propose architectural optimization based on individual user's preferences. By including the human factor in the abstraction layer of computer architecture, we have produced performance

enhancement and power consumption reduction in a computer system that is not possible otherwise.

System performance is typically quantified using that can be measured using metrics that are derived from low-level knowledge such as instruction throughput, hardware utilization. These metrics serve the role of proxies for user. However, every individual perceives system performance differently. Our PICSEL project was *pioneering* in estimating user-perceived performance using proxies (changes in display) that sit closer to the 'body' as compared to the 'metal'.

The beauty of the proposed holistic approach is the consideration of materials, the lowest layer present in computing system as well as the users, the highest layer. As technology scales further, variations in manufacturing technology has become prominent. The PDVS approach is one of the possible alternatives to utilize process variation to the advantage of the consumer. Recent industry trends in power management [119] for modern microprocessors support our assumption.

We questioned the basic assumption about hard constraint on system reliability. Our work on Clumsy Processors was the *first* to propose a system that violates the assumption that a circuit should work flawlessly even at the worst case scenario. We improved overall system performance and energy consumption by trading off reliability. The microarchitecture strategy for the next generation is called resilient microarchitecture that continually detects errors, isolates faults, confines faults, reconfigures the hardware, and thus adapts. If we can make such a

strategy work, there is no need for one-time factory testing or burn-in, since the system is capable of testing and reconfiguring itself to make itself work reliably throughout its lifetime. Clumsy Processing is one of the first steps towards such a resilient architecture.

Our work on intelligent task allocation [22] based on statistical nature of networking module processing was novel due its consideration for variability in processing time. Variation in execution time is an inherent property of any modular task. We can use a similar approach to solve the generic problem of task allocation in Chip Multiprocessors or any other scenario where tasks need to be distributed among a number of processing resources. We can adapt to similar approaches to achieve performance enhancement in any domain where data parallelism is present. Application domains exhibiting modular nature (e.g., data mining) may largely benefit from the proposed techniques.

Ideally a computing system should be heterogeneous so that it can support the wide variety of platforms, networks and services. It should be capable of adapting to user preferences, ensure correctness in a variable environment and allow optimized performance in a reconfigurable environment. The holistic vision of system architecture would provide an efficient solution to this multifaceted requirement.

We believe that satisfaction of the user is the prime objective of any kind of automation. Typically the surroundings of a user can vary from a resource-rich

environment (working with workstations) to resource-constraint settings (using a smart-phone). The generic applicability of holistic architecture can be implemented to this whole gamut of environment. Its philosophy is applicable on different application domains – embedded systems, networking hardware, high performance computing, to name a few. We feel the introduction of such hybrid architecture can benefit the whole population of computing systems. The new generation of autonomic system needs to fulfill two major constraints – fault-tolerance and fidelity-awareness. A fault tolerant system needs to detect and recover from fault for to meet the correctness objective. On the other hand, a fidelity-aware computing ensures the system can perform optimally with variability in available resources (CPU performance, power, network bandwidth, memory space). We have explored novel microarchitecture techniques that improve system performance through optimizations at every abstraction level (user, application, operating system, assembler, firmware, hardware and materials). We believe the philosophy of holistic computing architecture would be one of the most effective tools in the design of next generation computing system.

If you found this work interesting, and have additional questions, please contact me.

Arindam Mallik
arindam@eecs.northwestern.edu

REFERENCES

[1]. Brayton, R.K., G.D. Hatchtel, and A. Sangiovanni-VincenteUi, *A survey of optimization techniques for integrated circuit design.* IEEE Proceedings, 1981. 69(10): p. 1334-1362.

[2]. Otten, R.H.M. *Efficient Floorplan Optimization.* in *Proceedings of The International Conference on Computer Aided Design.* 1983.

[3]. Brooks, D., V. Tiwari, and M. Martonosi, *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*, in *ISCA*. 2000.

[4]. Brooks, D.M., P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P.W. Cook, *Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors.* IEEE Micro, 2000. 20(6): p. 26-44.

[5]. Austin, T., *DIVA: A Dynamic Approach to Microprocessor Verification.* Journal of Instruction Level Parallelism, May 2000. 2(11).

[6]. Austin, T. *DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design.* in *International Symposium on Microarchitecture*. Nov. 1999.

[7]. Ernst, D., N.S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. *Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation.* in *International Symposium on Microarchitecture*. Dec. 2003.

[8]. Ernst, D., S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N.S. Kim, and K. Flautner, *Razor: Circuit-Level Correction of Timing Errors for Low-Power Operation.* IEEE Micro, November/December, 2004. 24(6): p. 10-20.

[9]. Claasen, T., *The Changing Semiconductor Industry: From Components to Silicon Systems.* euromicro, 1999. 1: p. 1008.

[10]. Tanenbaum, A.S., *Structured Computer Organization*. 1979, Englewood Cliffs, New Jersey: Prentice-Hall.

[11]. Flynn, M.J., P. Hung, and K.W. Rudd, *Deep-Submicron Microprocessor Design Issues.* IEEE Micro, 1999. 19(4): p. 11-22.

[12]. De, V. and S. Borkar. *Technology and design challenges for low power and high performance*. in *Proceedings of the 1999 international symposium on Low power electronics and design*. 1999. San Diego, California, United States.

[13]. Borkar, S. *Microarchitecture and Design Challenges for Gigascale Integration - Keynote address*. in *The 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*. 2004.

[14]. Borkar, S. *Thousand Core Chips - A Technology Perspective*. in *44th ACM/IEEE Design Automation Conference, 2007. DAC '07*. 2007.

[15]. Iyer, R.K. *TRUSTED ILLIAC: A Configurable Hardware Framework for a Trusted Computing Base*. in *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium(HASE '07)*. 2007.

[16]. Cvetanovic, Z. and D. Bhandarkar. *Performance Characterization of the Alpha 21164 Microprocessor Using TP and SPEC Workloads.* in *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*. 1996. Washington, DC, USA.

[17]. Pancake, C., *The ubiquitous beauty of user-aware software.* ACM Communications Journal, 2001. 44(3).

[18]. Lee, C.B. and A.E. Snavely. *Precise and realistic utility functions for user-centric performance analysis of schedulers.* in *Proceedings of the 16th international symposium on High performance distributed computing (HPDC '07)*. 2007. Monterey, California, USA: ACM.

[19]. Mallik, A. and G. Memik. *A Case for Clumsy Packet Processors.* in *International Symposium on Microarchitecture*. Dec. 2004. Portland, OR.

[20]. Mallik, A., M.C. Wildrick, and G. Memik, *Application-Level Error Measurements for Network Processors* Institute of Electronics, Information and Communication Engineers (IEICE) Transactions on Information and Systems, 2005. E88-D(8): p. 1870-1877.

[21]. Mallik, A., M.C. Wildrick, and G. Memik. *Measuring Application Error Rates for Network Processors.* in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. July 2004. Hiroshima, Japan.

[22]. Mallik, A. and G. Memik. *Automated Task Distribution in Multicore Network Processors using Statistical Analysis*. in *The Symposium on Architectures for Networking and Communications Systems (ANCS-2007)* 2007.

[23]. Mallik, A., B. Lin, P. Dinda, G. Memik, and R.P. Dick, *Process and User Driven Dynamic Voltage and Frequency Scaling*, in *Technical Report NWU-EECS-06-11*. 2006, Department of Electrical Engineering and Computer Science, Northwestern University.

[24]. Dinda, P., G. Memik, R. Dick, B. Lin, A. Mallik, A. Gupta, and S. Rossoff. *The User In Experimental Computer Systems Research*. in *Workshop on Experimental Computer Sciencein conjunction with The Federated Computer Research Conference (FCRC) (submitted for review)*. 2007.

[25]. Srinivasan, G.R., *Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview*. IBM Journal of Research and Development, Jan. 1996. 40(1): p. 77-89.

[26]. HP. *Nonstop Computing*. [cited; Available from: http://nonstop.compaq.com].

[27]. Memik, G., W.H. Mangione-Smith, and W. Hu. *NetBench: A Benchmarking Suite for Network Processors*. in *International Conference on Computer-Aided Design (ICCAD)*. Nov. 2001. San Jose / CA.

[28]. Shivakumar, P., M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*. in *International Conference on Dependable Systems and Networks (DSN)*. June 2002.

[29]. Srinivasan Jayant, S.V.A., Pradip Bose , Jude A. Rivers. *The Impact of Technology Scaling on Lifetime Reliability*. in *DSN 04*. June 2004.

[30]. Shivakumar, P., M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*. in *2002 International Conference on Dependable Systems and Networks*. 2002: 389-398.

[31]. Hazucha P, S.C., *Impact of cmos technology scaling on the atmospheric neutron soft error rate.* IEEE Transactions on Nuclear Science, 2000. 47(6).

[32]. Karnik T., B.B., Soumyanath K, De V., Borkar S. *Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18u*. in *Symposium on VLSI Circuits*. 2001.

[33]. Seifert N., M.D., Leland N., Hokinson R. *Historical trend in alpha-particle induced soft error rates of the alphatm microprocessor*. in *39th Annual IEEE International Reliability Physics Symposium*. 2001.

[34]. International, O.f.S., *ISO Information Processing Systems - Data Communication High-Level Data Link Control Procedure - Frame Structure*. Oct. 1984.

[35]. Cell, R.R. *CRC-32 Calculation, Test Cases and HEC Tutorial*. [cited; Available from: http://cell.onecall.net/cell-relay/publications/software/].

[36]. FreeBSD, P.T., *FreeBSD Operating System*.

[37]. Baker, F., *Requirements for IP version 4 routers*. June 1995.

[38]. Shreedhar, M. and G. Varghese. *Efficient Fair Queuing using Deficit Round Robin*. in *SIGCOMM'95*. Aug/Sep 1995. Camridge / MA.

[39]. Rivest, R., *The MD5 Message-Digest Algorithm.* Apr. 1992.

[40]. RSA Data Security, I. *RSA Security Downloads.* [cited; Available from: http://www.rsasecurity.com/download].

[41]. PMC-Sierra, I. *URL-based Switching, PMC-2002232.* Feb. 2001 [cited; Available from: http://www.pmcsierra.com].

[42]. Burger, D. and T. Austin, *SimpleScalar Tool Set, Version 2.0.* June 1997, University of Wisconsin.

[43]. Turmon, M., R. Granat, and D. Katz. *Software-implemented fault detection for high-performance space applications.* in *International Conference on Dependable Systems and Networks (DSN).* June 2000.

[44]. Bose, P. *Ensuring dependable processor performance: an experience report on pre-silicon performance validation.* in *International Conference on Dependable Systems and Networks (DSN).* July 2000.

[45]. Anghel, L. and M. Nicolaidis. *Cost Reduction and Evaluation of a Temporary Faults Detecting Technique.* in *Design Automation and Test in Europe (DATE).* March 2000.

[46]. Patel, S.J., Z. Kalbarczyk, R.K. Iyer, W. Magda, and N. Nakka. *A Processor-Level Framework for High-Performance and High-Dependability.* in *Workshop on Evaluating and Architecting Systems for Dependability.* 2001.

[47]. Iyer, R.K. *Experimental Evaluation.* in *Special Issue of Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25).* 1995. Pasadena, CA.

[48]. Czeck, E. and D. Siewiorek, *Observations on the Effects of Fault Manifestation as a Function of Workload.* IEEE Transactions on Computers, May 1992. 41(5): p. 559--566.

[49]. Folkesson, P., S. Svensson, and J. Karlsson. *A Comparison of Simulation Based and Scan Chain Implemented Fault Injection.* in *28th Int. Symp. on Fault-Tolerant Computing (FTCS-28).* June 1998. Munich, Germany.

[50]. Borkar, S. *Thousand Core ChipsA Technology Perspective.* in *44th ACM/IEEE Design Automation Conference, 2007. DAC '07.* 2007.

[51]. Srinivasan, J.R., *Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview.* IBM Journal of Research and Development, Jan. 1996. 40(1): p. p. 77-89.

[52]. HP, *Nonstop computing, http://nonstop.compaq.com.*

[53]. Hamming, R.W., *Error Detecting and Correcting Codes.* Bell System Technical Journal, 1950. 26(2): p. 147-160.

[54]. Krishna, C.M. and L.-H. Lee. *Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time Systems.* in *Real Time Technology and Applications Symp.* May 2000.

[55]. Shivakumar, P., M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic.* in *International Conference on Dependable Systems and Networks (DSN).* June 2002.

[56]. Li, L., V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. *Soft error and energy consumption interactions: a data cache perspective.* in *ACM/IEEE International Symposium on Low Power Electronics and Design.* 2004.

[57]. Montanaro, J., R.T. Witek, K. Anne, A.J. Black, E.M. Cooper, D.W. Dobberpuhl, P.M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T.H. Lee, P.C.M. Lin, L. Madden, D. Murray, M.H. Pearce, S. Santhanam, K.J. Snyder, R. Stehpany, and S.C. Thierauf, *A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor.* IEEE Journal of Solid-State Circuits, 1996. 31(11): p. 1703-14.

[58]. Wilton, S. and N. Jouppi, *An enhanced access and cycle time model for on-chip caches.* July 1995, Digital Western Research Laboratory, 93/5.

[59]. Phelan, R., *Addressing Soft Errors in ARM Core-based SoC.* Dec. 2003, ARM Ltd.

[60]. Turmon, M., R. Granat, and D. Katz. *Software-implemented fault detection for high-performance space applications.* in *International Conference on Dependable Systems and Networks (DSN).* June 2000.

[61]. Srinivasan, G.R., *Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview.* IBM Journal of Research and Development, Jan. 1996. 40(1): p. p. 77-89.

[62]. Bose, P. *Ensuring dependable processor performance: an experience report on pre-silicon performance validation.* in *International Conference on Dependable Systems and Networks (DSN).* July 2000.

[63]. Anghel, L.a.M.N. *Cost Reduction and Evaluation of a Temporary Faults Detecting Technique*. in *Design Automation and Test in Europe (DATE)*. March 2000.

[64]. Annavaram, M., J.M. Patel, and E.S. Davidson. *Data prefetching by dependence graph precomputation*. in *28th Annual International Symposium on Computer Architecture*. 2001. Göteborg, Sweden.

[65]. Reinhardt, S.K. and S.S. Mukherjee. *Transient Fault Detection via Simultaneous Multithreading*. in *27th Annual International Symposium on Computer Architecture*. June 2000.

[66]. Mukherjee, S.S., M. Kontz, and S.K. Reinhardt. *Detailed Design and Evaluation of Redundant Multithreading Alternatives*. in *International Symposium on Computer Architecture (ISCA)*. May 2002.

[67]. Rashid, F., K. K. Saluja, and P. Ramanathan. *Fault tolerance through re-execution in multiscalar architecture*. in *International Conference on Dependable Systems and Networks (DSN)*. June 2000.

[68]. Gomaa, M., C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. *Transient-Fault Recovery for Chip Multiprocessors*. in *International Symposium on Computer Architecture*. June 2003. San Diego, CA.

[69]. Iyer, R.K. *Experimental Evaluation*. in *25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*. 1995. Pasadena, CA.

[70]. Mukherjee, S.S., C.T. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. *A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-*

Performance Microprocessor. in *International Symposium on Microarchitecture*. Dec. 2003.

[71]. Intel. *Intel® IXP2850 Network Processor.* 2004 [cited; Available from: http://www.intel.com/design/network/products/npfamily/ixp2850.htm].

[72]. Intel, C., *Intel® IXP2800 Network Processor Product Brief*. 2002: Santa Clara/CA.

[73]. Kohler, E., R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, *The Click modular router.* ACM Transactions on Computer Systems, 2000. 18(3): p. 263-97.

[74]. Shah, N., W. Plishker, and K. Keutzer. *NP-Click: A Programming Model for the Intel IXP1200.* in *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9).* February, 2003. Anaheim, CA.

[75]. Vin, H.M., J. Mudigonda, J. Jason, E.J. Johnson, R. Ju, A. Kunze, and R. Lian. *A Programming Environment for Packet-processing Systems: Design Considerations*. in *The Workshop on Network Processors & Applications - NP3. Held in conjunction with The 10th International Symposium on High-Performance Computer Architecture* 2004.

[76]. Intel, *Intel Microengine C Compiler Support: Reference Manual.* 2002.

[77]. Leary, K. and W. Waddington. *DSP/C: A Standard high level language for DSP and Numeric Processing.* in *Proc. Int. Conf. Acoustics, Speech and Signal Processing*. 1990.

[78]. Kahn, G. *The semantics of a simple language for parallel programming.* in *Proc. of the IFIP Congress 74*. 1974. North Holland.

[79]. Balarin, F., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. 1997, Massachusetts: Kluwer Academic Publisher.

[80]. Edwards, S. *Compiling Esterel into Sequential Code*. in *Proceedings of the 37th Design Automation Conference (DAC 2000)*. 2000. Los Angeles, California.

[81]. Wind River Systems Inc, *VxWorks Reference Manual*. 1999.

[82]. J. Nickolls et al, *Broadcom Calisto: A Multi-Channel Multi-Service Communications Platform*, in *Hot Chips*. 2002.

[83]. Baker, F., *Requirements for IP version 4 routers.* RFC 1812, June 1995.

[84]. Tsai, M., C. Kulkarni, C. Sauer, N. Shah, and K. Keutzer. *A Benchmarking Methodology for Network Processors*. in *1st Network Processor Workshop, 8th Int. Symposium on High Performance Architectures*. 2002.

[85]. Postel, J., *Internet Control Message Protocol. RFC 792 (Sept.), Internet Engineering Task Force. ftp://ftp.ietf.org/rfc/rfc0792.txt.* 1981.

[86]. Postel, J., *Internet Protocol. RFC 791 (Sept.), Internet Engineering Task Force.* ftp://ftp.ietf.org/rfc/rfc0791.txt, 1981.

[87]. Schreedhar, M. and G. Varghese. *Efficient Fair Queueing using Deficit Round Robin.* in *SIGCOMM'95*. Aug/Sep 1995. Cambridge, MA.

[88]. Kohler, E. *The Click Modular Router Project.* in *http://pdos.csail.mit.edu/click.*

[89]. Burger, D. and T. Austin, *The SimpleScalar Tool Set, Version 2.0.* 1997, Univ. of Wisconsin-Madison, Comp. Sc. Dept.

[90]. Devadas, S. and A.R. Newton., *Algorithms for Hardware Allocation in Datapath Synthesis.* IEEE Trans. On CAD, July 1989. 8, No. 7, pp. 768-781,(7).

[91]. Chekuri, C., *Approximation Algorithms for Scheduling Problems,Technical Report CS-TR-98-1611,*, Computer Science Department, Stanford University. August 1998.

[92]. Shachnai, H. and T. Tamir. *Polynomial time approximation schemes for class-constrained packing problems.* in *Proceedings of Workshop on Approximation Algorithms.* 2000.

[93]. Chen, M.K., X.F. Li, R. Lian, J.H. Lin, L. Liu, T. Liu, and R. Ju, *Shangri-La: achieving high performance from compiled network applications while enabling ease of programming.* ACM SIGPLAN Notices, 2005. 40(6): p. 224-236.

[94]. Gordon, M.I., W. Thies, and S. Amarasinghe, *Exploiting coarse-grained task, data, and pipeline parallelism in stream programs*, in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 2006. p. 151-162.

[95]. Plishker, W., K. Ravindran, N. Shah, and K. Keutzer. *Automated Task Allocation for Network Processors.* in *Network System Design Conference Proceedings.* October, 2004.

[96]. Srinivasan, A., *Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas.* Network Processor Design:Issues and Practices, November 2003.

[97]. Memik, G. and W.H. Mangione-Smith. *NEPAL: A Framework for Efficiently Structuring Applications for Network Processors.* in *Workshop on Network Processors – NP2 (held in conjunction with HPCA)*. Feb. 2003. Anaheim, CA.

[98]. Datar, S. and M.A. Franklin, *Task Scheduling of Processor Pipelines with Application to Network Processors*, Department of Computer Science and Engineering, Washington University in St.Louis.

[99]. Borkar, S., T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, *Parameter Variations and Impact on Circuits and Microarchitecture*, in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 2003.

[100]. Gupta, A., B. Lin, and P.A. Dinda, *Measuring and Understanding User Comfort with Resource Borrowing.* Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004), 2004.

[101]. Lin, B. and P. Dinda, *Putting the user in direct Control of CPU Scheduling.* The 15th IEEE International Symposium on High Performance Distributed Computing (HPDC), 2006.

[102]. Corporation, M. *Performance Logs and Alerts overview.* [cited; Available from: http://www.microsoft.com/windows2000/en/advanced/help].

[103]. Podien, W. *CPUCool.* [cited; Available from: http:// www. cpufsb. de/ CPUCOOL. HTM].

[104]. Wang, Z. and J. Crowcroft, *Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm.* ACM Computer Communications Review, 1992.

[105]. Stevens, W., *TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms.* Internet RFC 2001, 1997.

[106]. Fall, K. and S. Floyd, *Simulation-based comparisons of Tahoe, Reno and SACK TCP.* SIGCOMM Computer Communication Review, 1996. 26(3): p. 5-21.

[107]. Waizman, A. and C. Chung, *Resonant free Power Network Design using Extended Adaptive Voltage Positioning ({EAVP}) Methodology.* IEEE Transactions on Advanced Packaging, 2001. 24(3): p. 236-244.

[108]. Intel Corporation, *Intel Pentium M Datasheet.* [cited; Available from: http://developer.intel.com/design/mobile/pentium-m/documentation.htm].

[109]. Intel Corporation, *Intel Pentium M Processor Thermal Management.*

[110]. Jaider, M. *Notebook Hardware Control Personal Edition.* [cited; Available from: http://www.pbus-167. com/chc.htm].

[111]. Srinivasan, J., S.V. Adve, P. Bose, and J.A. Rivers. *The Case for Lifetime Reliability-Aware Microprocessors*. in *International Symposium on Computer Architecture (ISCA)*. June 2004. Munich, Germany.

[112]. Brock, B. and K. Rajamani. *Dynamic Power Management for Embedded Systems*. in *Proceedings of the IEEE SOC Conference*. 2003. Portland, Oregon, USA.

[113]. Gochman, S. and R. Ronen, *The Intel Pentium M Processor: Microarchitecture and Performance.* Intel Technology Journal, 2003.

[114]. Fei, Y., L. Zhong, and N.K. Jha, *An Energy-aware Framework for Coordinated Dynamic Software Management in Mobile Computers.* IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2004.

[115]. Choi, K., R. Soma, and M. Pedram, *Dynamic Voltage and Frequency Scaling based on Workload Decomposition.* Proceedings of The 2004 International Symposium on Low Power Electronics and Design (ISLPED '04), 2004: p. 174-179.

[116]. Wu, Q., V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D.W. Clark, *Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance.* 38th International Symposium on Microarchitecture (MICRO-38), 2005.

[117]. Ernst, D., N.S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, *Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation.* ACM/IEEE International Symposium on Microarchitecture (MICRO), 2003.

[118]. Dhar, S., D. Maksimovic, and B. Kranzen, *ClosedLoop Adaptive Voltage Scaling Controller For Standard Cell ASICs.* Proceedings of The International Symposium on Low Power Electronics and Design (ISLPED) 2005: p. 251-254.

[119]. John Wei. *Foxton Technology Pushes Processor Frequency, Application Performance.*

[120]. Lorch, J. and A. Smith, *Using User Interface Event Information in Dynamic Voltage Scaling Algorithms.* Technical Report UCB/CSD-02-1190, Computer Science Division, EECS, University of California at Berkeley, August 2002.

[121]. Yan, L., L. Zhong, and N.K. Jha, *User-perceived Latency based Dynamic Voltage Scaling for Interactive Applications.* Proceedings of ACM/IEEE Design Automation Conference (DAC '05), 2005.

[122]. Flautner, K. and T. Mudge, *Vertigo: Automatic Performance-Setting for Linux.* Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), 2002.

[123]. Xu, R., D. Moss, and R. Melhem, *Minimizing Expected Energy in Real-time Embedded Systems.* Proceedings of the 5th ACM international conference on Embedded software(EMSOFT), 2005: p. 251-254.

[124]. Gurun, S. and C. Krintz, *AutoDVS: an Automatic, General-purpose, Dynamic Clock Scheduling System for Hand-held Devices.* EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software, 2005: p. 218-226.

[125]. Ranganathan, P., E. Geelhoed, M. Manahan, and K. Nicholas, *Energy-Aware User Interfaces and Energy-Adaptive Displays.* Computer, 2006. 39(3): p. 31-38.

[126]. Anand, M., E. Nightingale, and J. Flinn, *Self-tuning Wireless Network Power Management*, in *The Ninth Annual International Conference on Mobile Computing and Networking (MobiCom'03).* 2003: San Diego, California, USA.

[127]. Cohen, A., F. Finkelstein, A. Mendelson, R. Ronen, and D. Rudoy, *On Estimating Optimal Performance of CPU Dynamic Thermal Management.* IEEE Computer Architecture Letters, 2003. 2(1).

[128]. Skadron, K., M.R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, *Temperature-aware Microarchitecture: Modeling and Implementation.* ACM Transactions Architecture Code Optimization, 2004. 1(1): p. 94-125.

[129]. Rohou, E. and M. Smith, *Dynamically Managing Processor Temperature and Power*, in *2nd Workshop on Feedback Directed Optimization*. 1999.

[130]. Liu, D. and C. Svensson, *Trading Speed for Low Power by Choice of Supply and Threshold Voltages.* IEEE Journal on Solid-State Circuits, 1993. 28: p. 10-17.

[131]. Brooks, D. and M. Martonosi, *Adaptive Thermal Management for High-Performance Microprocessors*, in *Workshop on Complexity Effective Design*. 2000.

[132]. Corporation, T., *The Technology Behind the Crusoe Processor*. 2000.

[133]. Ghinea, G. and J.P. Thomas, *Quality of Perception: User Quality of Service in Multimedia Presentations.* IEEE Transactions on Multimedia, 2005. 7(4): p. 786-789.

[134]. Gulliver, S.R. and G. Ghinea, *The Perceptual and Attentive Impact of Delay and Jitter in Multimedia Delivery.* IEEE Transactions on Broadcasting, 2007. 53(2): p. 449-458.

[135]. Wolfram Podien. *CPUCool*. [cited 2007; Available from: http://www.cpu-cool.de/index.html].

[136]. Corporation, M., *Windows Native Processor Performance Control*, in *Windows Platform Design Notes*. 2002, Microsoft Corporation.

[137]. Mallik, A., B. Lin, G. Memik, P. Dinda, and R.P. Dick, *User-Driven Frequency Scaling.* IEEE Computer Architecture Letters, 2006. 5(2): p. 16.

[138]. Wijesekera, D., J. Srivasyava, A. Nerode, and M. Foresti, *Experimental evaluation of loss perception in continuous media.* Multimedia Systems, 1999. 7(6): p. 486-499.

[139]. Claypool, M., K. Claypool, and F. Damaa, *The Effects of Frame Rate and Resolution on Users Playing First person Shooter Games*, in *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN) Conference*. 2006: San Jose, California, USA.

[140]. Mallik, A., B. Lin, G. Memik, P. Dinda, and R.P. Dick, *User-Driven Frequency Scaling.* IEEE Computer Architecture Letters (CAL), 2007.

[141]. Lin, B., A. Mallik, G. Memik, P. Dinda, and R.P. Dick. *Power Reduction Through Measurement and Modeling of Users and CPUs.* in *The International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS 2007).* 2007. California, USA.