Application aware performance, power consumption, and reliability tradeoff

Naga Pavan Kumar Gorti
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd
Part of the Computer Engineering Commons

Recommended Citation

This Dissertation is brought to you for free and open access by the Graduate College at Digital Repository @ Iowa State University. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Application aware performance, power consumption, and reliability tradeoff

by

Naga Pavan Kumar Gorti

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Arun K. Somani, Major Professor
    Akhilesh Tyagi
    Joseph Zambreno
    Philip Jones
    David Fernandez Baca

Iowa State University
Ames, Iowa
2014

Copyright © Naga Pavan Kumar Gorti, 2014. All rights reserved.
DEDICATION

To my family and to my daughter, who is yet to step into this beautiful world
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>xi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xiii</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Current trends in computing industry</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Performance trend</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Power consumption trend</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3 Reliability trend</td>
<td>3</td>
</tr>
<tr>
<td>1.1.4 Nature of PPR demands</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Research contributions</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Thesis organization</td>
<td>8</td>
</tr>
<tr>
<td>CHAPTER 2. REVIEW OF LITERATURE</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Dynamic voltage and frequency scaling</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Benefit of DVFS</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Early work on DVFS and classification of DVFS schemes</td>
<td>11</td>
</tr>
<tr>
<td>2.2.3 Online DVFS schemes</td>
<td>12</td>
</tr>
<tr>
<td>2.2.4 Offline DVFS schemes</td>
<td>14</td>
</tr>
<tr>
<td>2.2.5 Hybrid DVFS schemes</td>
<td>15</td>
</tr>
<tr>
<td>2.2.6 Thermal aware DVFS</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Microarchitectural adaptation</td>
<td>18</td>
</tr>
</tbody>
</table>
2.3.1 Classic research in microarchitectural adaptation .......................... 19
2.3.2 Closely related research in microarchitectural adaptation ............... 26
2.4 Uniqueness of current research ...................................................... 29

CHAPTER 3. PERFORMANCE RELIABILITY TRADEOFF ....................... 30
3.1 Introduction ..................................................................................... 30
3.2 System model .................................................................................. 31
  3.2.1 Assumptions ............................................................................... 34
  3.2.2 Procurement of task execution characteristics ............................... 34
3.3 OC Selection ................................................................................... 36
  3.3.1 GOPS Algorithms ..................................................................... 37
  3.3.2 LOPS Algorithm ....................................................................... 43
3.4 Evaluation of the developed DVFS based schemes ............................. 44
  3.4.1 Experimentation with synthetic task sets ..................................... 45
  3.4.2 Simulation based performance reliability tradeoff analysis .......... 50
3.5 Performance reliability tradeoff using DVFS and microarchitectural adaptation 56
  3.5.1 Need for considering DVFS and microarchitectural adaptation together 56
  3.5.2 Selection of adaptive microarchitectural components .................. 57
  3.5.3 Performance reliability tradeoff considering both DVFS and microarchitectural adaptation .................. 59
3.6 Conclusion ....................................................................................... 60

CHAPTER 4. ADAPTIVE MICROARCHITECTURAL CONFIGURATION
SPACE PRUNING ................................................................. 61
4.1 Introduction ..................................................................................... 61
4.2 Selection of Advantageous Control Knobs (SACK) ............................ 63
  4.2.1 Observations ............................................................................. 64
4.3 Elimination of ineffective configurations (ELIC) ............................. 66
  4.3.1 Observations ............................................................................. 66
4.4 Configuration Set Selection for Runtime (CSSR) ............................ 67
4.4.1 Merit based selection ........................................... 68
4.4.2 Bound based selection ........................................ 70
4.4.3 Neighborhood based selection ............................... 73
4.5 Evaluation of the different CSSR pruning methods ........... 76
  4.5.1 Final configuration space ................................. 76
  4.5.2 User demand tracking ................................... 76
4.6 Conclusion .................................................. 79

CHAPTER 5. DEGRADATION OF PERFORMANCE-POWER TRADE-OFF UNDER PERMANENT FAULTS .................. 82
  5.1 Introduction .................................................. 82
  5.2 Fault model .................................................. 83
  5.3 Evaluation of tradeoff degradation ......................... 87
    5.3.1 Dispatch port failure ................................ 87
    5.3.2 Cache way failure ................................... 89
    5.3.3 Instruction window chunk failure .................... 90
    5.3.4 Voltage and frequency control failure ............... 91
    5.3.5 Power saving with reduced performance requirements 95
    5.3.6 Avoiding available adaptations for increased power saving 96
  5.4 Conclusion .................................................. 98

CHAPTER 6. APPLICATION AWARE PERFORMANCE-POWER TRADE-OFF .................................................. 100
  6.1 Introduction .................................................. 100
  6.2 Two stage static cum dynamic adaptation strategy .......... 102
    6.2.1 Application phase demarcation ....................... 102
    6.2.2 Static reconfiguration stage ......................... 104
    6.2.3 Dynamic reconfiguration stage ....................... 108
  6.3 Dynamic adaptation strategies ............................ 110
  6.4 Evaluation .................................................. 112
6.4.1 Evaluation Methodology ............................................. 112
6.4.2 Determination of maximum interval length .................. 114
6.4.3 Adaptation strategies .............................................. 116
6.4.4 Handling runtime variations in performance and power .......... 117
6.4.5 Comparison of SDC and dynamic adaptation strategies ........... 118
6.4.6 Scaling of power consumption with performance .................. 121
6.4.7 Comparison with previous schemes .............................. 122
6.5 Conclusion ........................................................... 124

CHAPTER 7. CONCLUSIONS AND FUTURE WORK ..................... 126
7.1 Conclusions ........................................................ 126
7.2 Future work ......................................................... 128
LIST OF TABLES

Table 3.1 OC selection schemes for Peak Reduction ....................... 39
Table 3.2 Evaluation parameters used for analyzing effectiveness of GOPS algorithms ............................................... 46
Table 3.3 QOS satisfaction and Energy savings .............................. 49
Table 3.4 Simulation parameters used for performance-reliability tradeoff analysis .................................................... 51
Table 3.5 SPEC workloads used for simulations ............................. 52
Table 3.6 MTTF model parameters ............................................. 54
Table 3.7 Operating voltages and frequencies for Intel Pentium M processor .. 56
Table 3.8 Adaptive hardware configurations .................................. 58
Table 4.1 Considered adaptive components and adaptations ................ 64
Table 4.2 tp for the considered adaptive components ........................ 65
Table 4.3 Performance-power variations provided using the chosen configuration space ..................................................... 65
Table 4.4 Different pruning methods for CSSR ................................ 68
Table 4.5 Different performance-power demand scenarios .................. 77
Table 5.1 Investigated fault scenarios .......................................... 86
Table 5.2 Performance and power characteristics obtained for different performance demands .......................... 95
## LIST OF FIGURES

| Figure 1.1 | Transistor count trend (per chip) for commercial processors | 2 |
| Figure 1.2 | Core count trend for HPC platforms | 2 |
| Figure 1.3 | Power consumption trend for HPC platforms | 3 |
| Figure 1.4 | Relative importance of factors limiting server growth potential | 4 |
| Figure 1.5 | Decrease in lifetime reliability of processors with shrinking gate length | 5 |
| Figure 2.1 | Adaptation in Complexity-Adaptive processors | 20 |
| Figure 2.2 | *Smart Memories* tile floorplan | 25 |
| Figure 2.3 | 2Bc-gskew-pskew branch predictor organization | 26 |
| Figure 3.1 | Example Operations Table | 33 |
| Figure 3.2 | Example OC selection using WOPS | 42 |
| Figure 3.3 | Effectiveness of GOPS algorithms in reducing inter-task temperature gradients | 47 |
| Figure 3.4 | Effectiveness of GOPS algorithms in providing energy savings | 48 |
| Figure 3.5 | Scaling of the number of algorithmic iterations of GOPS algorithms with task set size and task stretch factor | 49 |
| Figure 3.6 | Simulation framework | 50 |
| Figure 3.7 | MTTF increase using DVFS using (a) Window based selection, and (b) Peak reduction | 55 |
| Figure 3.8 | Normalized performance vs. (a) IL1-Assoc. (b) Operating VF for selected SPEC benchmarks | 57 |
| Figure 3.9 | Normalized power vs. (a) IL1-Assoc. (b) Operating VF for selected SPEC benchmarks | 57 |
Figure 3.10 Power consumption breakdown among different units on Alpha EV6 floorplan .............................................................. 58
Figure 3.11 Expected MTTF improvement through the combined use of DVFS and microarchitectural adaptation with (a) Window based selection, and (b) Peak reduction ............................................ 59
Figure 4.1 Number of configurations eliminated by ELIC .......................................................... 67
Figure 4.2 Usage frequency of the individual adaptive settings for the considered adaptive components ........................................ 68
Figure 4.3 Example merit based selection for CSSR. n=8 and k=4 ............................................ 70
Figure 4.4 CSSR using bound-based selection when n=8 and k=5 ............................................ 72
Figure 4.5 Graphical representation of the retained configuration space for soplex benchmark after ELIC pruning step ........................................ 73
Figure 4.6 Final adaptive microarchitectural configuration space size ........................................ 77
Figure 4.7 PI in tracking high performance demands .............................................................. 78
Figure 4.8 PI in tracking low power demands .............................................................. 78
Figure 4.9 PI in tracking balanced demands .............................................................. 79
Figure 4.10 PI in tracking stringent demands .............................................................. 80
Figure 5.1 Tradeoff degradation when one dispatch port fails ............................................ 88
Figure 5.2 Tradeoff degradation when one cache way fails ............................................ 90
Figure 5.3 Tradeoff degradation when an instruction window chunk fails ........................................ 91
Figure 5.4 Tradeoff degradation when the lowest VF setting fails ........................................ 92
Figure 5.5 Tradeoff degradation when the intermediate VF setting fails ........................................ 93
Figure 5.6 Tradeoff degradation when the highest VF setting fails ........................................ 94
Figure 5.7 Deliverable peak performance and the associated power consumption utilizing a subset of available configuration space ........................................ 97
Figure 6.1 Adaptive architectural reconfiguration process ............................................ 104
Figure 6.2 Evaluation platform .............................................................. 113
Figure 6.3  Phase count scaling with maximum phase length  . . . . . . . . . . . . 115
Figure 6.4  PI degradation with maximum phase length  . . . . . . . . . . . . 115
Figure 6.5  Benefit of utilizing intra-application adaptation  . . . . . . . . . . . . 116
Figure 6.6  Effectiveness of static cum dynamic adaptation strategy  . . . . . . . . 118
Figure 6.7  Comparison of PI for SDC and dynamic adaptation strategies  . . . 119
Figure 6.8  Performance delivered for various performance demands  . . . . . . 121
Figure 6.9  Power consumed for various performance demands  . . . . . . . . . . 122
Figure 6.10 Performance delivered for peak power constraint of 70%  . . . . . . 123
Figure 6.11 Energy efficiency when serving different performance needs  . . . . 124
Figure 7.1  Normalized performance vs. normalized power for FFT  . . . . . . 129
Figure 7.2  Normalized performance vs. normalized power for Barnes hut algorithm 130
I feel glad in taking this opportunity to express my gratitude to people who helped me with my graduate education and research in particular and life in general.

Firstly, I would like to thank Dr. Arun K. Somani for being my major professor. He has been the guiding beacon for my research. I have always been inspired by his discipline, dedication and intelligence. I would like to specially thank him for his constant effort to impart good presentation and technical writing skills to me, and for encouraging free thinking.

I would also like to thank Dr. Akhilesh Tyagi, Dr. Joseph Zambreno, Dr. Philip Jones, and Dr. David Fernandes-Baca for serving on my committee and providing me with constructive feedback regarding my research. I have also been associated with all these people in a classroom setting and am thankful to them for honing my thinking abilities that proved essential for my research. I am also thankful to other professors in the ECpE department: Dr. Manimaran, Dr. Zhao Zhang, Dr. Nicola Elia and Dr. Chris Chu for teaching me various aspects of computing and optimization. I would like to extend a special thanks to the ECpE department and Iowa State University for providing me an excellent atmosphere to conduct my research.

Research is often inspired by the daily conversations we have with other people. My research group at Iowa State University, namely the Dependable Computing and Networking Laboratory, enabled me to network with a large set of such people. I would like to specially thank Prem, Prasad, Nishanth, Vishwanathan, David, Parijat, Karthik, Piyush, Cory, Ashish, Joy, Utkarsh, Haoyuan, Teng, Jin Xu, Koray, Matt, Lizandro, Harini, Suresh, Ganesh, and Kritanjali among others in this regard.

I express my heartfelt gratitude to my parents and grandparents for teaching me about life and pushing me towards academic success. I am very fortunate to meet the love of my life, Priyam, at Iowa State University. It would be very fair to say that I would not have accomplished what I have in my research without her continuous support. I am extremely
indebted to my uncle and his family who helped me both in academic and non-academic aspects of my life. I would also like extend a special thanks to my cousin Karthik for taking care of my transition from India to USA and all the help he has rendered to me. Finally, I would like to thank my dear friends Sharath, Adwait, Teja, Ramya, Jyani, Shantan, Indranil, Madhu, Yuzhu, Sista, Azhar, Swagath, Avinash, Harsha, Guru, Suman, and many others for the wonderful times they have shared with me and made my journey through PhD very pleasant.
ABSTRACT

There has been an unprecedented increase in the drive for microprocessor performance. This drive is motivated by the increase in software complexity, opportunity to solve previously unattempted problems especially in scientific domain, and a need to crunch the ever growing Big Data to enable a multitude of technological advances to benefit mankind. A consequence of these phenomena is the ever increasing transistor count in deployed computing systems.

Although technology scaling leads to lower power consumption per transistor, the overall system level power consumption is on the rise. This leads to a variety of power supply related issues. As the chip die area is not increasing significantly, and the supply voltage reduction is not keeping on par with the reduction in device dimensions, an increase in power density is observed. This manifests as an increased temperature profile on the chip floorplan. A rise in temperature necessitates aggressive and costly cooling mechanisms adding to the design complexity and manufacturing efforts. It also triggers various failure mechanisms leading to reduction in the expected chip lifetime/reliability.

Given the conflicting trends in Performance, Power consumption, and chip Reliability (PPR), it is imperative to balance them in a fine-grained fashion to meet system level goals and expectations. Sole dependence on the advancements in manufacturing technology is no longer sufficient. Alternate venues for PPR management are being increasingly paid attention to.

On the other hand, the PPR demands are usually time dependent. For example, the constraint on power consumption in a green data center is dictated by the energy reserve. The demand on performance in a cloud based platform depends on the agreed Quality of Service (QOS) requirements. The reliability of a microprocessor is dependent on the deployment domain.

The goal of our research is to address the issue of growing microprocessor power consumption subject to performance and/or reliability goals. Through our developed schemes, we tailor the
execution context to match application requirements. This leads to judicious use of power while adhering to aforementioned constraints. It is to be noted that the actual demands on performance, power consumption, and reliability are highly variant, and depend upon executing applications and operating conditions. As such, we develop schemes to cater to these variant demands.

To meet these demands efficiently, the solutions developed are tailored to current hardware-software interaction characteristics. Two techniques that are very relevant in this area, namely dynamic voltage and frequency scaling (DVFS) and microarchitectural adaptation, are leveraged to produce expected PPR characteristics when executing a wide variety of tasks.

In this dissertation, we demonstrate how the expected chip lifetime can be augmented in a real-time setting using DVFS while paying heed to performance constraints modeled as QoS requirements. Individual tasks in a task queue are assigned specific voltage and frequency pairs to utilize for their execution. This assignment is empowered by knowledge of application-wise hardware-software interactions to reach solutions that are tailored to the current execution scenario. Our observations indicate that a 2 to 18 fold improvement in chip lifetime can be expected by the utilization of the schemes we develop in this regard. Capitalizing on the power of microarchitectural adaptation, we further improve chip lifetime expectations 2-8 times, based on the failure mechanism investigated. This increase in expected chip lifetime directly translates to reduction of both operational and replacement costs.

We also provide mechanisms to co-manage performance and power consumption constraints. Comprehensive microarchitectural adaptation space is very complex and its usage thus leads to significant runtime overhead. To tackle this, we devote a fair bit of attention to its pruning so as to narrow down on and utilize only the most effective adaptations. A two stage adaptation process is provided to a) improve optimality of the solutions delivered, and b) to keep the runtime overhead in check. We observe that our schemes provide 20% higher normalized energy efficiency compared to the state of the art techniques proposed, while using just a very small fraction of the configuration space. We also find that our schemes effectively cater to a wide variety of demands on performance and power consumption, providing the necessary hardware characteristics within 10% bound.
Since only the most useful configuration space is retained for adaptation, occurrence of a fault that prohibits the usage of a certain adaptive control can lead to the inability to satisfy a subset of hardware demands. A detailed analysis has been carried out to understand how the remaining active configurations can preserve the expected hardware behavior. To a good extent, we observe that the system behavior under a failure closely tracks (with less than 5% tracking error) the obtainable behavior without the presence of the fault.

We believe that application tailored schemes for PPR management become increasingly relevant as the microprocessor design advancements saturate in the future. They will be extremely relevant to extract every possible ounce of performance while confirming to constraints on power consumption and reliability. Given the effectiveness of our schemes, we are confident that such schemes are applicable in different markets like embedded computing, desktop computing, cloud platforms and high performance computing. Insights drawn from our research will guide chip designers in the provision of effective adaptive controls to combat increasing demands on PPR characteristics.
CHAPTER 1. INTRODUCTION

This chapter discusses some recent trends in computing industry and their implications on microprocessor characteristics. The need for managing performance, power consumption, and/or reliability ($PPR$) through careful hardware-software co-design is detailed. This is followed by outlining the contributions of the current research. An overview of how the thesis is organized is presented at the end of this chapter.

1.1 Current trends in computing industry

1.1.1 Performance trend

Demand for computing performance is growing at an unprecedented pace. This demand is motivated by several factors. First, the increase in software complexity necessitates aggressive hardware designs to provide acceptable latency, response time or throughput. Second, scientific computing community is attempting to solve ever challenging large problems at high speeds. Examples of such problems include genome sequencing, weather modeling, molecular dynamics simulations, etc. Quick processing of applications in this domain results in major advances in our understanding of the universe and everything it encompasses. Third, immense computing potential is required to handle 'big data' that is being produced in various fields like social networking, universities, remote sensing, etc. It is reported by IBM [46] that more than 90% data in this world is generated within the past two years.

To satisfy the growing drive for performance, more and more transistors are crammed onto computing platforms. The increased transistor count is used to both $scale$-$in$ and $scale$-$out$ the computing architectures. $Scale$-$in$ refers to increasing the chip complexity by making cores more and more aggressive. $Scale$-$out$ refers to the increase in number of cores employed in a
compute node as well as number of compute nodes employed in a computing platform. Figure 1.1 [104] and 1.2 [97] show the consequences of scale-in and scale-out respectively in industry over the last 14 years. In particular, we can observe that the industry has been consistently outperforming Moore’s law based predictions starting from 2000.

### 1.1.2 Power consumption trend

According to semiconductor scaling theory [4], the power consumption per transistor decreases by a factor of $U^2$ with each new technology generation, where $U$ is the reduction factor for supply voltage. However, the trends in transistor count, especially due to scale-out, pave way for increased net power consumption. This occurs despite the growing maturity in the semiconductor manufacturing and device scaling. Figure 1.3 [97] shows the increase in system level power consumption for the most power hungry HPC platforms over the past 14 years.
It is to be noted that this increase can be observed in the desktop and embedded computing segment as well. Figure 1.3 serves just as an example showing the trend in one particular computing segment.

The increased power consumption leads to various issues related to power supply and inflates electricity bills in the high capacity server domain. According to a survey by Intel [47], the challenges involved in powering and cooling constitute the number one factor limiting the expansion of server industry to meet the current global demand. The survey’s results depicted in Figure 1.4 shows that 59% of the surveyed people concur that power consumption is the bottleneck in the development of server market. In the embedded and personal computing domain, the increase in power consumption leads to decreased battery life and discomfort in device handling. It also necessitates the design of aggressive cooling mechanisms and expensive heat sinks [101, 88, 70].

1.1.3 Reliability trend

We have noted that the power consumption of a chip reduces by a factor of $U^2$ for each new technology generation, where $U$ is the voltage scaling factor. It is also the case that the transistor density scales up by a factor of $S^2$, where $S$ is the device dimension scaling factor. It is a well-known fact that $U < S$ because a) departure from standardized voltage levels raises compatibility issues, and b) decrease in supply voltage leads to leakage power due to the corresponding decrease in the threshold voltage $V_{th}$ needed to maintain proper noise margins. As such, the power density scales as $S^2/U^2$. This ratio is greater than 1. Due to
the increased power density, the temperature at which the chip operates increases as well. An increase in temperature reduces the lifetime of the devices. In fact, the expected chip lifetime goes down by a factor of 2 as the operating temperature rises by $10^0 \degree C$ [78]. Although most of the power dissipated as heat can be effectively discarded from the chip packaging by heat sinks, this will not hold well in the future given the rate of rise in chip power density. Active cooling mechanisms like fans are not possible for computing platforms like smartphones and tablets. Some aggressive and innovative cooling techniques like fluid based cooling have been implemented in the high end server market (ex. IBM Aquasar [108]). However, the usage of such mechanisms for low end desktop and embedded computing markets is far-fetched.

It is be noticed that the expected lifetime for a processor depends on the application executing on it, in addition to the technology dependent parameters. This is because the various applications exercise the different units on the chip floorplan to varying degrees, leading to different operating temperatures. Figure 1.5 [92] shows how the lifetime of processors changes in accordance with transistor gate length and the executing applications (from SPEC 2000 suite [38]). The lifetime is modeled in terms of chip Mean Time To Failure (MTTF).

In addition to decrease in lifetime reliability, there are a few other issues with increasing chip temperature. Firstly, the carrier mobility decreases with increasing temperature, leading to performance degradation. Secondly, the sub-threshold leakage, which is a dominant leakage power mechanism increases with increasing temperature. The increased thermal energy
Figure 1.5 Decrease in lifetime reliability of processors with shrinking gate length

possessed by the electrons due to increased temperature makes it easy to traverse the channel when gate voltages are lower than the threshold. Thirdly, the interconnect delay increases since its resistivity increases with temperature.

From the trends discussed so far, it is clear that PPR considerations play a major role in both the design and operation of computing infrastructure. It is not sufficient to optimize the hardware to satisfy one of the PPR demands at a time. Increasing performance generally leads to higher power consumption and decreased reliability. As such, it is important to co-manage performance-power and performance-reliability together. This dissertation makes an effort to provide efficient solutions to the aforementioned co-management issues.

1.1.4 Nature of PPR demands

Computational needs in terms of delivered PPR characteristics are often varied over time. In the server domain, high performance is required when the system load is high and the agents loading the system are guaranteed a high Quality of Service (QoS). Performance can be traded off for lower power consumption and higher reliability during periods of low activity. Servers employed in green data centers are powered by electricity generated through various renewable energy sources, like sun and wind. Since the availability of these sources is time variant, so will the power generated be. In spite of maintaining energy reservoirs, it becomes imperative to
look for measures to tradeoff performance with power to avoid catastrophic blackouts. In the personal and embedded computing domain, one might require high performance when executing intense tasks, while low power operation may be required when running on battery without any active power sources. A major motivation to reduce power consumption in these domains is the impracticality of designing active cooling mechanisms like fans and to avoid the use of aggressive heat sinks to keep costs low.

It is clear from the $PPR$ trends discussed in the last subsection that sole reliance on technology scaling to tackle the related issues is not sufficient. Exploration of alternate venues to manage all or a few of $PPR$ demands is crucial. It is also the case that rigid architectural decisions to provide expected $PPR$ characteristics is impossible at microprocessor design time. For example, we have seen in Figure 1.5 that reliability of a processor is dependent on the applications executing on it. Similar arguments can also be made for performance and power consumption. As such, a single architectural configuration does not lead to similar $PPR$ characteristics when executing different applications.

We strongly believe that the solution to the intricate problem of providing required $PPR$ characteristics can be achieved through careful hardware-software co-design. Our research leverages on the simple fact that applications vary with regards to their interaction with hardware. As a result, different hardware components become critical to provide good performance for different applications. Leveraging this fact, our research focuses on utilization of $DVFS$ and microarchitectural adaptation hand in hand to produce expected behavior on a microprocessor when executing a variety of applications. The complexity and effectiveness of such techniques in $PPR$ management are analyzed and tackled. The techniques we develop in this research work utilize static knowledge of hardware-software interactions to provide good performance and a given power level or to provide good reliability at a given performance level. Simplistic runtime mechanisms are also developed to ascertain rigid $PPR$ guarantees in the face of misjudgment of the static expectations and runtime uncertainties.
1.2 Research contributions

The goal of this dissertation is to address the issue of growing microprocessor power consumption subject to performance and/or reliability constraints. We provide application-aware mechanisms for PPR management according to the corresponding constraints specified as inputs. It is clear that provision of good performance is orthogonal to lowering power consumption or improving reliability. As such, a design goal is to balance out the obtainable PPR characteristics in the most efficient manner in accordance with the priority associated with the different constraints. Since at least a component of the management mechanisms falls under application execution time, we include the design goal to make such a component fast and lightweight in nature. The main contributions of this thesis are as follows.

1. Existing DVFS schemes are not equipped to deal with thermal cycling. This phenomenon occurs when the chip temperature rapidly fluctuates due to the varying hardware-software interactions resulting from different applications executing sequentially. We include this awareness into DVFS mechanism and observe an improved expectation in chip lifetime.

2. Microarchitectural adaptation space is very large, given the number of adaptive components and the ways in which each component can be individually configured. We make a conscious effort to methodically prune the configuration space to retain only the most effective configurations for adaptation. We evaluate how our pruned configuration space can be used to produce varied PPR characteristics.

3. Static schemes for PPR management are comprehensive and provide the most optimal operation characteristics. However, they suffer on account of their rigidity and inability to adapt to runtime fluctuations. On the other hand, dynamic schemes are adaptive but they are not optimal due to runtime overhead considerations. We combine the advantages of these two approaches to develop hybrid schemes that contain a comprehensive static component to optimize the decision process as well as a runtime component to adapt to runtime fluctuations. We also develop alternate lightweight runtime only schemes that perform close to the comprehensive approaches.
4. We utilize simplistic metrics as inputs reflecting the PPR demands/expectations. This helps users with varied computer literacy levels to be able to specify and achieve required characteristics from hardware.

1.3 Thesis organization

The reminder of this thesis is organized as follows. Chapter 2 provides an overview of previous research on both DVFS and microarchitectural adaptation. A few examples of classical work as well as current state-of-the-art approaches in these areas are elaborated in detail. Chapter 3 describes how DVFS alone as well as DVFS in conjunction with microarchitectural adaptation can lead to increased chip reliability in real-time environments. A Quality of Service (QoS) metric is used to dictate performance requirement. A methodology to prune adaptive microarchitectural configuration space is presented in Chapter 4. Since we reduce the configuration space to include only the most beneficial adaptations for performance-power tradeoff, Chapter 5 analyzes how the tradeoff is affected when a single configuration out of this set is unavailable for use due to a permanent fault. Chapter 6 deals with application aware performance-power tradeoff. Questions on when and how to adapt the microarchitecture to satisfy set constraints on performance and power are answered in this chapter. The merit of using the developed adaptation mechanisms in catering to varying user demands from hardware is investigated and the relevant results are presented. We conclude our discussion in Chapter 7 by reiterating the achievements of the research work. Since the research is focused on a uniprocessor adaptation, a discussion on how to leverage this work to perform adaptations in homogeneous or heterogeneous multicore environments is added. The conclusions of this discussion will trigger investigation of multicore adaptation.
CHAPTER 2. REVIEW OF LITERATURE

This chapter provides an overview of previous research on both DVFS and microarchitectural adaptation. A few examples of classical work in this area are elaborated in detail. Also, some cutting-edge approaches that are proposed recently are discussed and distinguished. We conclude this chapter by specifying how the current research work differs from the existing body of work.

2.1 Introduction

The drive for improved microprocessor performance led to aggressive superscalar designs during late 90’s and early 2000’s. These superscalar designs increased in their complexity from being a simplistic single-issue low frequency design to multi-issue designs operating at very high frequencies. As a result, the chip power consumption rose exponentially, needing aggressive heat sink designs to adequately dissipate the heat produced. The increased power consumption motivated the research for a variety of hardware/software based techniques to counter the increased power dissipation. An alternate solution proposed was to design and utilize multicores. Due to the increasing aggressiveness of $PPR$ constraints, there exists a need to adopt both these solutions together. In this dissertation, we focus on the hardware/software based techniques for $PPR$ management or tradeoff.

A large number of schemes were developed to manage power and/or limit energy consumption and/or maintain hardware reliability, while sacrificing little performance. These schemes are based either on microarchitectural adaptation or $DVFS$. The primary objective of these schemes can be one of the following: 1) reduction of dynamic power, 2) reduction of static power, 3) adhering to a particular power or energy budget, etc. The hardware entity carefully
regulated to achieve the set objective can be the CPU voltage and/or frequency, or the various microarchitectural entities like issue queue size, number of active functional units, certain ways of cache, etc. Some schemes are particularly targeted towards multi-cores while others are generic and can be applied individually to each core. In the following sections, we will look at a few classic as well as state-of-the-art schemes based on DVFS and microarchitectural adaptation before distinguishing the current research from the previous work. Since there exist a plethora of schemes in both these areas, we limit ourselves to providing examples of ground breaking or closely related work.

2.2 Dynamic voltage and frequency scaling

2.2.1 Benefit of DVFS

The power (both static-$P_{\text{static}}$ and dynamic-$P_{\text{dynamic}}$) consumed by an application executing on a processor depends on the core voltage ($V$) and frequency ($f$) as

\[
P_{\text{static}} \propto V
\]

\[
P_{\text{dynamic}} \propto V^2 f
\]

Hence, lowering the core voltage and frequency is an effective way to manage power consumption, and indirectly, the chip reliability. When $V$ is lowered, the circuits become slower since it takes more time to charge and discharge the load capacitances in the CMOS gates. As such, $f$ has to be lowered to allow enough time to charge a capacitance to reflect a logic '1' or to discharge it to reflect a '0'. Reduction in voltage and frequency negatively affects system performance as well. It is well known that the time consumed to execute an application ($t$) can be computed by

\[
t = \frac{\text{instruction\_count} \times CPI}{f}
\]

where $\text{instruction\_count}$ and $CPI$ refer to the number of instructions in the program and the average cycles per instruction respectively. From the above equation, the time consumed is inversely proportional to core frequency. However, the $CPI$ changes with frequency since
memory latency is unaffected by core frequency change. Hence, a sub-linear decrease in performance can be observed with lowered voltage and frequency. These trends strongly motivate the use of DVFS for PPR management.

2.2.2 Early work on DVFS and classification of DVFS schemes

One of the earliest research works proposing the reduction in voltage to counter power consumption was put forward by Chandrakasan et al [16]. To reduce the detrimental effect on performance, pipelining and resource duplication were proposed. Using performance and power consumption modeling techniques, the authors derive the optimal voltage and frequency to minimize energy-delay product. However, voltage and frequency are not dynamically varied during execution.

Since then, a slew of DVFS techniques have been proposed for general purpose processors (eg. [81, 77]), embedded systems (eg. [80, 53, 76]), high performance computing platforms (eg. [42, 40, 57, 60]), as well as real-time systems (eg. [80, 107, 18]). Real-time systems have the notion of task deadlines which can be used as explicit performance constraints. In the other computing domains, such a notion does not exist. Hence, a number of alternate metrics are targeted for optimization like energy efficiency, energy-delay product, etc.

DVFS incurs two kinds of performance overhead. As stated earlier, the decrease in voltage leads to slower circuits and hence, the operable frequency. In addition, the transitions between different voltage and frequency levels incur runtime overhead as well as the time taken for making the deployed voltage and frequency decisions. As such, any DVFS scheme should avoid very frequent voltage and frequency transitions and has to be able to make decisions rather quickly.

Based upon when and how the deployed voltage and/or frequency is (re-)assigned, existing schemes in this field can be classified into offline [50, 72, 85, 43, 66], online [102, 34, 20, 73, 28, 94], and hybrid schemes [75, 90]. In an offline scheme, the decisions on when and how to perform DVFS are taken statically. The decisions are usually based on expected application execution profiles or code characteristics. Since decisions are taken statically, more complex analysis can be performed leading to better DVFS decisions. However, they cannot exploit
runtime hardware-software interactions. In an online scheme, all the DVFS related decisions are based upon observed system state and a few hardware-software interaction metrics traced during execution. These schemes are lightweight in nature and work on peepholes of instruction traces profiled at runtime. Hybrid schemes combine the benefits of both the online and offline schemes.

2.2.3 Online DVFS schemes

In these schemes, DVFS trigger points can be interval-based (window-based) or event based. Weiser et al. [102] put forward the idea of interval-based DVFS for general purpose computing domain. The authors used trace based simulations to evaluate a set of DVFS approaches. The traces used for simulations were collected from a Unix based workstation over a period of many hours. Three different DVFS schemes were considered.

1. OPT: The entire execution trace is analyzed and the runtime for all the tasks are stretched to fill all the idle times.

2. FUTURE: Similar to OPT, but peers into a small window in future rather than the entire trace. Runtime for the tasks within the window are stretched.

3. PAST: A small window in the past execution profile is considered, and decisions for a future window are based on hardware-software interactions observed in the previous window.

The authors observed that the merits of FUTURE and OPT schemes approach that of the OPT scheme when the DVFS intervals or window sizes are lengthened.

Govil et al. [34] proposed a few additional sophisticated schemes for DVFS management. Two new metrics are recoded for each interval and utilized for their schemes. The first metric, namely \textit{run\_percent}, computes the percentage of time for which the CPU is active. The second metric, namely \textit{excess\_cycles} represents the work in an interval that is not accomplished by using the selected speed setting. The new schemes proposed base the voltage and frequency decisions on being able to run the work corresponding to a predicted \textit{run\_percent} for the current
interval as well as the $\text{excess}_{cycles}$ accumulated over the previous intervals. Different methods are used to predict the $\text{run}_{percent}$ for the current interval. The different prediction schemes are as follows.

1. FLAT: A speed setting, which can accomplish the predicted work for the current interval plus the $\text{excess}_{cycles}$ pushed into the current interval is selected. The $\text{run}_{percent}$ is assumed to be a constant.

2. LONG\_SHORT: Two averages of previous $\text{run}_{percents}$, one averaged over the last 3 intervals and the other averaged over the last 12 intervals is maintained. The $\text{run}_{percent}$ for the current interval is predicted as a weighted average of the previous two recordings. A speed setting that satisfies the predicted $\text{run}_{percent}$ and the accumulated $\text{excess}_{cycles}$ is selected for the current interval.

3. AGED\_AVERAGES: This is a variant of LONG\_SHORT, where the predicted $\text{run}_{percent}$ is calculated by a weighted average as in the previous case. The weights decrease as we go deeper into the past.

4. CYCLE: A cyclic behavior of $\text{run}_{percents}$ in the past history is looked for. If such a cycle is found, it is extended to predict the $\text{run}_{percent}$ for the current interval. If not, a constant $\text{run}_{percent}$ is assumed.

5. PATTERN: The $\text{run}_{percents}$ from previous intervals are categorized into course spells. The pattern of the $\text{run}_{percents}$ for the last few intervals is matched against patterns in the deeper past successively until a match occurs. The extension of such a pattern in the past dictates the predicted $\text{run}_{percent}$.

The authors observed that FLAT and LONG\_SHORT provide the best energy savings compared to the other strategies.

There are two problems inherent with interval driven schemes based on past workload traces. First, it may not be possible to always find a suitable pattern in the deeper past reflecting the characteristics associated with the profile in the recent past. This results in inaccurate workload
prediction for future intervals. Second, the assumption that future workload characteristics mirror the past may not hold well.

Childers et al. [20] proposed to use external demands on the target Millions of instructions per second (MIPS) to make voltage and frequency scaling decisions. New scaling decisions are made once for an interval size of 2µs. The new frequency for an interval \( f_{\text{new}} \) is calculated as

\[
f_{\text{new}} = f_{\text{old}} \times \frac{\text{MIPS}_{\text{goal}}}{\text{MIPS}_{\text{observed}}} \tag{2.4}
\]

where \( f_{\text{old}} \) corresponds to the frequency employed for the previous interval. \( \text{MIPS}_{\text{goal}} \) and \( \text{MIPS}_{\text{observed}} \) are the MIPS set as the target and the observed value for the previous interval respectively. The authors observed a 47% reduction in energy consumption using their scheme when compared to using a fixed voltage and frequency.

Dhiman et al. [28] propose a machine learning based approach for DVFS. A set of hardware-software interactions are recorded using hardware counters. These interactions are used to classify the current execution context into one of the regions/baskets in possible execution spectrum. Each basket has associated with it a voltage and frequency range that is deployed. The authors achieved a 49% decrease in energy consumption and reduce the implementation overhead by a factor of 2 over existing state of the art approaches.

Event driven DVFS schemes trigger a voltage and scaling decision when a certain hardware event occurs. For example, Marculescu [73] used the knowledge of CPU stall cycles to switch the processor to lower voltage and frequency levels while preserving performance. The author observed a 20% reduction in energy consumption, 22% reduction in power consumption, and 14% reduction in peak power while sacrificing less than 6% performance. Stanley et al. [94] similarly propose hardware based monitoring to detect application regions that are memory bound and reduce voltage and frequency to reduce energy consumption.

### 2.2.4 Offline DVFS schemes

Offline DVFS algorithms calculate the trigger points and deployed voltage and frequency levels statically. Such algorithms calculate the optimal voltages and frequencies by formulating and solving an ILP based upon the set constraints. Ishihara and Yasuura [50] provided an
ILP formulation for selecting optimum voltage levels for a task execution given a processor with discrete voltage levels. They provided an additional insight that employing just two voltage levels per task can optimize the energy consumption. In contrast, the authors of [72] provided a solution to the optimal voltage and frequency selection problem with the assumption of continuous voltage and frequency levels. We believe that the consideration of continuous voltages and frequencies is impractical and Ishihara and Yasuura’s approach can also cater to such situations, if they exist. Zomaya et al. [85] made similar observations for using just two discrete frequency levels capitalizing on dynamic frequency scaling.

A compiler level DVFS technique was proposed by Hsu and Kremer [43] where the compiler instruments applications to supply DVFS related commands. The application is profiled to identify regions in application code with differing timing characteristics and execution frequencies. For each program region, an optimal voltage and frequency is assigned so that the performance penalty is never more than a preset threshold. Since the DVFS decisions are based on profiled characteristics, they would not hold tight with future executions.

Offline techniques for DVFS are comprehensive and result in optimum system behavior. However, the solutions are not adaptive to runtime fluctuations in expected performance and/or power consumption characteristics. On the other hand, online techniques can adjust to runtime deviations in application behavior. Since their operation falls under actual runtime, they have to be lightweight and hence consider only intervals of execution for making voltage and frequency selection decisions. As such, they are non-optimal. Hybrid schemes, which leverage the advantages of both these techniques are highly desirable.

2.2.5 Hybrid DVFS schemes

Checkpointing based hybrid DVFS techniques are proposed by [75, 90]. In the scheme proposed in [90], static voltage and frequency assignments are provided for different intervals within an application’s worst case execution path. Intervals are demarcated at the branching edges of the control flow graph, which correspond to branch or loop statements. When the actual execution path deviated from the predicted path, the expected time difference for the application executing along these two paths is used to speed up or slow down the processor
accordingly. Also, the predicted execution path is now updated. Using their schemes resulted in 34% lower energy consumption when compared to the state of the art intra-task DVFS schemes while executing MPEG-4 decoder program.

**DVFS in HPC domain** Use of DVFS to provide energy savings in high performance computing domain has been proposed by [57] and [60]. The inter-task computational imbalance is used to slow down tasks on the non-critical path to achieve significant energy savings. Freeh et al. in [32] run the application to collect profile information. Using this information, the application is divided by hand into multiple phases. Once the phase boundaries are demarcated, the application is augmented to use the different voltage/frequency pairs to determine the best combination to use for each phase. The authors in [63] take this a step further by automating the phase boundary and voltage/frequency pair selection. The hit rates of the different caches, the ratio of number of floating point operations to the number of memory operations, etc. are used to characterize the behavior of individual loops. Each loop is then compared against a known set of benchmark loops in terms of these observed characteristics. Each benchmark loop has an associated voltage/frequency pair that is deemed best for it. The pair corresponding to the matched benchmark loop is utilized for a program loop.

### 2.2.6 Thermal aware DVFS

As technology scales down, the power density on the chip increases leading to higher temperatures. The increase in temperature necessitates the use of costly heat sinks and cooling mechanisms. To counter this demand and to preserve hardware reliability, thermal awareness has been introduced into the voltage and frequency selection process.

Xie et al. [106] proposed thermal aware task scheduling policies. These policies scheduled tasks on a System on chip (SOC) based upon a set of calculated static and dynamic criticalities. Application control flow graph is used to calculate the static criticality of tasks. Dynamic criticalities were based both on the position of a task in the control flow graph and the expected operating temperature. Although voltage and frequency scaling is not used, we mention this work since it represents one of the earliest efforts in thermal aware task scheduling.
Bao et al. [7] looked at temperature aware voltage selection. The authors propose a scheme that takes the task mapping onto a multicore SOC as an input. The target temperature at which a core should run also constitutes an input to the voltage selection process. First, a set of voltages are assigned to the tasks to minimize the energy consumption. The thermal profile produced as a result is used to readjust the voltages assigned to re-adjust the temperature profile. This process is repeated until the temperature converges to the set target. No particular constraints on performance are considered.

The authors of [17] propose a mixed ILP based approach to assigning and scheduling tasks with hard real-time constraints in MPSOCs. To solve large problem instances, heuristics are also proposed. The chip peak temperature is subject to constraints in this approach.

It is observed that rapid temperature variations on the chip, along with the absolute temperatures are responsible for a large number of chip failures [87]. This chip failure mechanism, called thermal cycling, has been studied in previous research but not dealt with proactively. For example, the authors of [24] show the effect of thermal cycles on lifetime reliability, but do not put forward an online approach to voltage scaling taking into account the thermal cycles. The scheme proposed by Bao et al. [7] can be used to tackle thermal cycling but performance awareness needs to be integrated into the scheme, essentially leading to a complete redesign. The authors in [26] develop an ILP based approach to tackle thermal challenges. Thermal cycles are minimized indirectly by constraining the peak temperature. This scheme is static and does not deal with runtime variations in temperature and expected performance whose knowledge is assumed for the ILP formulation. An online strategy is proposed in [25] where an intelligent runtime management system selects one of the possible thermal policies to manage temperature profiles. However, the authors do not consider specific performance bounds. Instead, performance and various reliability aspects are treated with the same priority. In this dissertation, we develop a set of schemes to co-manage performance and chip reliability (including proactive management of thermal cycles) employing DVFS. Specific bounds on performance are utilized and guaranteed as per defined Quality of Service (QoS) standards while improving reliability. Effects of both absolute temperatures and thermal gradients are considered in reliability calculations.
Sueur and Heiser [64] pointed out that the efficacy of DVFS alone in achieving energy savings is diminishing over time due to 1) the increase in leakage power component, 2) reduced memory latency, and 3) improved sleep modes. In fact, their experiments revealed that DVFS could result in increased energy usage on modern hardware platforms. As such, the benefits provided by DVFS should be coupled with techniques leading to leakage power reduction to provide an increased dynamic range of performance-power values obtainable on a given architecture. In the next section, a review of such a technique leveraging on the capability to tune the aggressiveness of the different microarchitectural components is provided.

2.3 Miroarchitectural adaptation

Previous work in this area can be classified according to the adaptation granularity measured along two directions. Spatial adaptivity corresponds to the size of configuration space considered. A larger configuration space leads to fine-grained control at the expense of analysis complexity and runtime overhead. Temporal adaptivity refers to the frequency with which adaptations are carried out during an application execution. Similar to the previous case, a tradeoff between control granularity and runtime overhead exists in this regard.

Researchers have proposed adaptive architectural schemes using a single component [1, 13, 33, 44, 100] or multiple components [2, 6, 31, 51, 61, 74]) adaptation. For a single component, the configuration space is small, leading to faster reconfiguration decisions. The authors of [100] proposed a cache where the fetch size is continually modified based on the application access patterns. The authors of [13] present a circuit design targeting issue queue in a superscalar processor where the speed and size are adapted. Branch target buffer adaptation is performed in [44] along with adapting the components of a hybrid predictor to save significant amount of energy consumption with very little performance loss. The authors of [1] present a methodology to deactivate certain cache ways based on application cache intensity. Some performance degradation is also tolerated in this process to attain considerable energy savings.

For multi-component schemes, the authors in [2] have adapted the L1 and L2 caches, reorder buffer, instruction and load/store queues and register files. Cache resizing and DVFS are considered by [74]. Very large configuration spaces are handled in [61, 31, 65] as well.
Microarchitectural adaptation schemes with different temporal granularity have been proposed. The authors in [71] use a single architectural configuration for an application. The configuration chosen can change for different applications. Such schemes are simplistic but do not exploit intra-application variations in hardware-software interactions. Dhodapkar et al. [30] base adaptation decisions on working set signatures. An application execution profile is split into different regions each having a particular execution signature. This signature is constructed from hardware-software interactions. All regions having a similar signature use the same adaptive configuration. Different configurations that are amenable to different application signatures are constructed and used. Adaptation based on frame level granularity is proposed for multimedia applications in [45]. Adaptations are also considered at the granularity of application intervals [65, 61].

In the following subsections, we will be reviewing a few important or closely related research works in microarchitectural adaptation.

2.3.1 Classic research in microarchitectural adaptation

**IPC and clock speed adaptation** The roots of research in microarchitectural adaptation can be traced back to the work of David Albonesi, who introduced the concept of Complexity-Adaptive processors (CAP) [1]. A CAP employed configurable hardware for the core superscalar control and cache resources. A dynamic clock is also provided to adapt along with the envisaged hardware structure to optimize the clock speed based on the inherent delay of the configurable hardware resources. Using these features, the Instructions committed per cycle (IPC) and clock speed can be traded off with one another. The idea behind the envisaged tradeoff is discussed below.

Superscalar microprocessor resources like cache hierarchy, branch predictor, register rename logic, instruction queue, issue logic, etc. are traditionally implemented as RAM or CAM based arrays implemented as replicated storage elements driven by global address and data buses, as shown in Figure 2.1 [1]. The inherent circuit delay associated with such resources is dependent on the number of active elements in the array. If a few elements in an array are deactivated, the inherent circuit delay is reduced, thereby making the circuit operable with a higher clock speed. Simultaneously, the IPC takes a hit. The
actual tradeoff potential is dependent on the application that is being executed and the RAM or CAM array that is being adapted. Based on this idea, Albonesi configured data caches and instruction queue to suit the needs of various applications.

**Cache line size adaptation**  
Veidenbaum et al. suggested adapting cache line sizes to suit application needs [100]. The line size of a 16 KB L1 cache has been adapted between the values of 8 and 256 bytes, each higher size double the size of the previous value. Since the line size adaption would require reconfiguration of circuitry between RAM and the cache under consideration, as well as some aspects of the RAM design, a virtual line size is defined and adapted. Instead of adapting the physical line size, which is assumed to be 8 bytes in this research, multiple cache lines (constituting a single virtual line) are transferred to the cache from RAM upon a cache miss.

The adaption scheme reduces the virtual line size upon a cache miss if the previously fetched words are not being used, but increases the size when an adjacent line is already present in the cache. In this context, the adjacent line for a particular virtual line is defined as the cache
line of same size which would have been part of the line under interest, had the virtual line size been double of the present value. To know the line usage and adjacency information, each physical line is augmented to store three extra pieces of information.

1. The current virtual line size corresponding to the physical line

2. A single adjacent bit, to indicate if the adjacent line is already present in the cache

3. A 2-bit saturating counter which measures the use of the cache line of interest

The speed of adaptation is further tuned by sending out line size decrement or increment requests upon a cache miss. A real adaptation occurs when a particular number of such requests occur consequently for a single cache line upon a cache miss associated with it. The authors considered a mixture of two adaptation approaches that are outlined below.

1. **Inc-fast**: Increment line size instantly but decrement only after 2 consecutive decrement requests are issued for a single line.

2. **Dec-fast**: Decrement line size instantly but increment only after 2 consecutive increment requests are issued for a single line.

The authors found that the most effective solution to reduce memory bandwidth was to use **inc-fast** for small line sizes and **dec-fast** for large line sizes. However, the miss rate encountered was observed to be higher for some benchmarks when compared to a statically determined best line width for the particular benchmark.

**Cache and TLB hierarchy adaptation**  A memory hierarchy reconfiguration scheme was proposed by Balasubramonian et al. [5]. In this work, a single large L1 cache is converted to a mixture of L1 and L2 caches, and a single large TLB is converted into a two-level TLB through runtime adaptation. The cache and TLB usage are monitored by detecting phase changes using miss rates and branch frequencies, and performance is boosted by balancing the hit latency and miss latency intolerance during execution.
The authors start out with a 2 MB 4-way data cache and convert it into a mixture of L1 and L2 caches using intelligent addition of repeater switches to electrically isolate specific wordlines of interest, upon L1 cache access. The following configurations for the L1 cache are allowed.

1. 256 KB directly mapped cache
2. 768 KB, 3-way cache
3. 1 MB, 4-way cache
4. 1.5 MB, 3-way cache
5. 2 MB, 4-way cache

The cache miss rate, IPC, and branch frequency are monitored for every 100K cycles of execution (called an interval) using hardware counters. When an application starts executing, the L1 cache is configured to be 256 KB by default and an optimal cache exploration process initializes. After every interval during this exploration process, the obtained CPI is recorded. If the cache miss rate is greater than 1%, the next larger L1 size is chosen for the next interval. This process continues until the largest cache size is selected, or the miss rate drops to a value less than 1%. At the end of this process, the configuration that provides the smallest CPI is chosen and is used for the future intervals of execution. Further optimal cache size explorations are necessitated when the number of branches and misses significantly differ beyond a set threshold between two intervals of execution. If successive exploration phases lead to the same optimal cache line size, the threshold is incremented. Otherwise, the threshold is decremented. This avoids unnecessary exploration phases where a single optimal cache configuration will be selected.

To adapt the TLB, a counter tracks TLB miss handler cycles for every 1 million cycles of execution. The L1 TLB size is incremented if this counter exceeds 3% of the total execution time for the interval. In contrast, the L1 TLB size is reduced if the TLB usage is less than half.

Application of this adaptation methodology for a two level cache and TLB hierarchy at 0.1µm technology led to an improvement of 15% CPI when compared to a best conventional two-level hierarchy of a comparable size. The authors also experimented with a conventional L1
cache and a similarly adapted L2/L3 cache in sub-0.1\( \mu \)m technology. In this case, the following energy aware modifications are also made.

1. Only low energy configurations are used for L2 cache.

2. Data and tag lookup processes are serialized.

Using these modifications, a reduction in 43% of energy consumed by the memory hierarchy is observed, in addition to the performance improvement.

**Microprocessor queue adaptation** Dmitry Ponomarev et al. proposed adaptation of the sizes of issue queue (IQ), the reorder buffer (ROB), and the load/store queue (LSQ) based on periodic sampling of their occupancies. These components are adapted independently of one another, and the interplay between adaptation of the various components is not considered. Further, these occupancies were not monitored continuously, to avoid overhead. All the queues are assumed to be designed using individually controllable partitions, each of which can be activated or deactivated using a simple control signal.

Different strategies are used to deactivate (downsize) or activate (upsize) partitions. The downsizing of partitions is considered periodically. During each period, the number of active entries in a queue are sampled at regular intervals. An average number of active entries is derived from this monitored data. At the end of a period, the difference between the current size of the queue and the average active queue size is calculated. If the difference is greater than the size of a single partition, downsizing occurs. Two different downsizing strategies are implemented and analyzed.

1. Conservative downsizing: Only one partition is deactivated at the maximum for a single period.

2. Aggressive downsizing: Multiple partitions, which fit within the difference calculated at the end of the period, are subject to deactivation.

An overflow counter is used for each individual queue to support upsizing. This counter measures the number of cycles the microprocessor pipeline is stalled because of inability to find an
empty slot in the queue. This counter is initialized to 0 upon commencement of application execution, and after each upsizing operation. An upsizing is associated with the overflow of this counter. Only a single partition is reactivated during each upsizing operation.

The authors evaluated the effectiveness of the proposed adaptation methodology by performing the aforementioned adaptations during the execution of various SPEC 95 benchmarks [84]. The Simplescalar simulator [12] is modified to provide the proposed adaptations and the execution of these benchmarks is simulated on a 4-way superscalar processor. During this course, an average of 53 power savings is observed for the combination of IQ, ROB, and LSQ, while incurring a performance penalty of only 5%.

**Modular reconfiguration** Mai et al. [71] proposed a high-level modular reconfiguration platform called *Smart Memories*. This platform combines the benefit of architectural adaptation with the performance advantage of domain specific computing hardware, while providing a streamlined modular architecture that can be reconfigured easily to match the demands of applications from multiple domains.

A *Smart Memories* chip consists of multiple processing tiles, each containing configurable memory, wiring, and processing resources which employ multiple computational models. Figure 2.2 shows the *Smart Memories* tile floorplan. The floorplan consists of a processor, local memory organized in multiple blocks called mats, and an interface to interact with other tiles in the system. A crossbar is provided for the processor to interact with the local memory, as well as other tiles in the system. To support multiprocessing, dedicated networking is provided between sets of 4 tiles each, which are referred to as a *Quad*. Per *Quad* DRAM resources are also provided to promote efficient communication. Microarchitectural adaptation support is provided for the processor and the local memory in each tile. The processor in each tile has two integer clusters and one floating point cluster to enable parallel execution. The *Smart Memories* instruction path can be configured to support wide or narrow instruction encoding. A 256-bit wide instruction format suits explicitly parallel instructions found in media and signal processing kernels. The processor used this wide instructions to supply instructions to all available units parallely. A 128-bit VLIW instruction format is supported to benefit
applications which contain ILP but are less regular. A 32-bit narrow instruction format is further supported to benefit applications that do not exhibit high ILP. However, thread-level parallelism is used for such applications, where parallel execution of 2 concurrent threads issuing 32-bit instructions is supported. Each Smart Memory tile mats can be configured in multiple ways to support different cache organizations. Further, memory accesses can be performed both in regular mode, or an auto-decrement/auto-increment mode with configurable strides.

To showcase the flexibility to the adapted hardware, two widely variant processor topologies, namely the Hydra speculative multiprocessor [35] and the Imagine stream processor [56], are mapped onto the Smart Memories architecture. Although these implementations perform sub-optimally with respect to their corresponding standalone implementations, the authors claim that the flexibility provided to map multiple architectures onto the proposed platform trumps the sub-optimality.

**Branch predictor adaptation** Huang et al. [44] proposed reconfiguring various branch predictor parameters through structure resizing and access gating to minimize energy consumption during periods of ineffective branch prediction. The authors tie the branch prediction accuracy to program code structure and utilize profiling to analyze branch prediction efficacy (of different allowed branch predictor configurations) individually for the different program
subroutines. Based on the data obtained, the components of a hybrid branch predictor are reconfigured at the granularity of program subroutines by solving a knapsack problem to minimize overall energy consumption. The program code is dynamically instrumented to reconfigure the branch predictor. This offline approach minimizes runtime overhead in determining the optimal branch predictor configuration. Figure 2.3 shows the baseline hybrid predictor configuration employed by the authors. The branch predictor contains three top level components: $gskew$, $pskew$, and $bimodal$. The $gskew$ and $pskew$ components can be activated/deactivated using the $GEN$ and $PEN$ signals respectively. The $bimodal$ component is not targeted for reconfiguration since it does not consume significant amount of energy. The number of sets in the branch target buffer are also adapted in addition to the $gskew$ and $pskew$ components.

2.3.2 Closely related research in microarchitectural adaptation

Reducing Peak Power with a Table-Driven Adaptive Processor Core Kontorinis et al. [61] present a processor peak power management technique. Peak power is of importance since it directly affects the thermal budgeting, packaging and cooling costs for the processor. The authors control the peak power by designing a centralized control mechanism that controls architectural configuration. Peak configuration is not assigned to all the adapted units at the same time. The units that are considered for adaptation are the I and D caches, integer and FP instruction queues, reorder buffer, load/ store units, integer and FP execution units, and register renaming unit. A table driven approach is utilized for consultation on power and performance characteristics before making a configuration decision.
Their design consists of two major components- A config ROM and an adaptation manager. For any given application, the config ROM is first loaded with a set of allowed configurations that do not surpass the peak power limits statically decided. The configurations once set stay constant for a long interval of time which they call epoch (1M instructions). During this epoch, the adaptation manager collects performance characteristics from the processor. At the end of an epoch, the adaptation manager chooses a new configuration from the config ROM based on the performance characteristics. Using this technique, the authors are able to reduce the peak power by 25% by sacrificing no more than 5% of performance.

This work draws some powerful conclusions and inferences. Firstly, the authors observe that setting just 2 or 3 microarchitectural units at their maximum level are enough to achieve high levels of performance. Secondly, by limiting the allowed peak power to appropriate levels, the search space for the configurations has been reduced drastically. For example, out of 6,144 configurations allowed by the permutations between the different configuration levels of the adaptive hardware units considered, only 285 combinations hold valid when the peak power limit is to be reduced by at least 15%.

There are two drawbacks with the proposed methodology. The size of the configuration search space is entirely dependent on the allowed peak power level. There is no interest to remove a subset of the combinations that perform very closely. As such, the pruning methodology is not very robust. Secondly, there is no analysis on how better the proposed techniques work can when the epoch length is varied. This would make a case of optimal control granularity.

**Predictive Model for Dynamic Microarchitectural Adaptivity Control** This research [31] presents a prediction based model to improve the energy efficiency of a processor. The energy efficiency is defined as the ratio of number of instruction executed to the energy consumed by the processor. The model developed is constructed empirically by identifying optimal designs on training data. The model takes as input a set of hardware characteristics monitored and predicts the best architectural configuration for the 14 adaptive units considered. The characteristics monitored are in the form of temporal counters with different bins for different ranges of values.
A soft-max distribution is assumed for the probability of the model generating a certain output configuration given a set of temporal hardware statistics. This distribution has certain parameters associated with it, which need to be quantified in order to use the model for generating optimal configurations later on. In order to obtain these parameters, the authors formulated a training process where a large number of program phases are studied against the corresponding optimal configurations.

During a real program execution, the hardware characteristics logged when the program enters a new phase of execution. These characteristics are then fed to the generated model which will output the optimal configuration for all the adaptive components. The authors found that their model is effective in doubling the energy efficiency.

Since the model uses the parameters obtained through working on a few selected applications, it may not be reflective of the requirements of any generic application executing on the processor. The authors do not justify the effectiveness of using the large set of adaptive controls, since each control potentially has a different degree of effect in improving energy efficiency. Also, the effect of configuring the different architectural controls are dealt with separately and their interplay is not paid attention to.

**Efficiency trends and limits from comprehensive microarchitectural adaptivity**

The large dimensionality of microarchitectural configuration space analysis prohibits designer from considering a wide range of adaptations since the corresponding analysis becomes prohibitively expensive. Two possible solutions exist. First, the analysis accuracy can be toned down by employing sampling and predictive modeling techniques. Second, the adaptive configuration space is reduced as per requirements using existing simulation methodologies. Lee and Brooks [65] employ the former strategy and consider an adaptive architectural space containing 240B configurations. The authors utilize random instruction trace sampling, spline based regression for predictive modeling, and genetic algorithms for refinement of considered configuration space. They also consider the ill effects of low degree of spatial and temporal adaptivity in delivering the optimal $\frac{bips}{w}$, which is their optimization metric. The authors also perform an analysis of how DVFS adds to system efficiency on top of microarchitectural adaptation.
However, this analysis is performed at a later stage and we believe that microarchitectural adaptation and DVFS considerations should go hand in hand rather than one after the other. In addition, we believe that reducing the microarchitectural adaptation configuration space to a small subset of possible adaptations will significantly decrease the control complexity as well as the design complexity. To this end, we develop mechanisms to systematically prune the available configuration space.

2.4 Uniqueness of current research

The following factors distinguish the current research from the existing body of work.

1. Existing DVFS based schemes for performance-reliability management do not focus on reduction of thermal cycling. We include the constraint to maintain the chip temperature within a small window while adhering to performance constraints. This translates to higher chip lifetime expectations, as observed in our experiments.

2. There have been no schemes that evaluate the effectiveness of microarchitectural adaptation for performance-reliability co-management with thermal cycling awareness. We show how addition of microarchitectural adaptation can improve chip lifetime expectations when combined with DVFS.

3. A few pointers are provided in the earlier research to reduce the adaptive configuration space. However, no formal pruning methodology has been proposed in this regard. We develop a three stage pruning methodology to 1) identify the most effective adaptive controls to build in hardware, and 2) choose the most beneficial configurations to use given an application. Such a pruning methodology is essential to decrease the design and control complexity associated with microarchitectural adaptation.

4. Existing schemes do not attempt to evaluate the effectiveness of the working adaptive configuration space in trading off performance and power in the presence of faults. Such an analysis is necessary and indicates what additional hardware capability is needed to handle failures gracefully. We carry out such an analysis in this research work.
CHAPTER 3. PERFORMANCE RELIABILITY TRADEOFF

This chapter presents details of our research on performance and reliability tradeoff. A soft real-time environment is considered for application execution. Performance guarantees are provided through a Quality of Service (QoS) constraint. The chip reliability is studied in terms of the expected Mean time to failure (MTTF) under different chip failure mechanisms. Specific hardware parameters are adapted to suit application needs and the constraint on performance, while improving expected chip lifetime. A two stage performance and reliability management scheme employing DVFS is designed and its effectiveness in providing the required tradeoff is investigated. This scheme is then augmented with microarchitectural adaptation to further improve the chip lifetime expectation.

3.1 Introduction

The need to manage performance and reliability together has been detailed in Chapter 1. Microprocessor performance and hardware reliability characteristics are generally orthogonal in nature. Increasing performance requires increasing the aggressiveness of hardware which leads to higher power density. This increased power density translates to higher temperature and reduced lifetime reliability. Since different applications stress the hardware to different degrees, higher performance goals also lead to larger thermal gradients. This further reduces the reliability. As such, it is important to provide mechanisms to cater to different demands representing these two entities. Since performance is generally of higher importance than reliability, we consider it as the primary constraint while reliability is considered as a secondary constraint. We develop a scheme to improve chip lifetime expectations when executing a set of known tasks while adhering to a performance constraint.
As mentioned earlier, heterogeneous tasks executing on a processor drive the chip to different temperatures. The various microarchitectural components are exercised to different extents, due to the varying CPU and memory intensities of these tasks. In [93], the authors report the chip peak temperatures when different SPEC2000 integer benchmarks are executed on processors from different technology nodes. Their report provides the insight that the temperature difference on a single processor when executing different benchmarks goes up as the feature size shrinks. Executing a series of tasks leading to different temperature profiles causes the processor to heat up and cool down alternatively, a phenomenon known as thermal cycling. Thermal cycles are categorized into large and small scale cycles. The large scale thermal cycles are often a consequence of switching the processor on and off, and are expected to happen infrequently. The small scale thermal cycles occur more frequently, and are a function of the executing task characteristics.

Electronic circuits essentially constitute interconnection of materials with different thermal coefficients (e.g. metals and dielectrics). These individual materials are subjected to differential expansion and contraction when the chip is heated and cooled. Due to their physical interconnection, a mechanical stress develops in this process, leading to die cracking, thin film cracking, solder joint fatigue, etc., over time. The authors of [87] mention that the corresponding failure mechanisms contribute to a large chunk of chip failures. Most of these failure mechanisms have not been analyzed statistically due to the complexity involved in the intricate interplay among the various mechanisms. Even the JEDEC testing standards [52], used for chip reliability quantification, do not account for the effects of small scale thermal cycles. We make an effort to reduce both the thermal cycles and the steady state temperatures (SSTs) in an integrated fashion, thereby increasing the expected chip lifetime. In the following, we refer to SST as a relatively constant temperature to which a chip is heated when executing an application.

3.2 System model

We choose a soft real-time system model for application scheduling and execution. Task schedulers in real-time environments typically base their schedule construction on the worst case execution times (WCET) of the tasks. When deadlines are not tight, static slack accumu-
lates in the schedule. The actual execution times also differ from the worst case assumptions since the application contains multiple execution paths, each incurring potentially different execution time. As a result, dynamic slack also accumulates in the schedule. Most modern processors typically support execution of the tasks at multiple frequencies (and correspondingly voltages). As a result, any positive slack generated in the system is currently exploited by lowering the operating voltage and frequency. This reduces the overall energy consumption and chip temperature. A majority of the existing DVFS approaches are not thermally aware and aggressively tune down the operating voltages (frequencies) to achieve maximal energy savings. To the best of our knowledge, there have been no approaches targeted to handle the small scale thermal cycles. The following system model is considered.

1. There are $N$ tasks in the system $(T_1, T_2, ..., T_N)$ in order of decreasing priority i.e., $Pri(T_i) > Pri(T_j)$ when $i<j$ and $Pri(T_k)$ represents the priority of task $T_k$. Such task lists are readily available in real-time environments. Such lists also exist in batch processing systems. It is further assumed that a non pre-emptive real-time scheduler is present which assigns priority to these tasks.

2. The underlying processor support $m$ modes of operation. Each mode $k$, represented by $c_k$, is associated with a specific voltage $v_k$ and frequency $f_k$, and optionally a unique hardware configuration. For the purposes of DVFS, $c_k$ contains just $v_k$ and $f_k$. In a later section, we vary the hardware configuration along with DVFS to improve reliability.

3. Each task $T_i$ is associated with a deadline $D_i$, an expected execution time $t_{ik}$, an expected energy consumption $E_{ik}$ (product of power consumption and execution time), and steady state temperature $T_{ik}$ for the $k^{th}$ mode of operation.

The different task characteristics mentioned above are arranged in a 2-Dimensional grid which we refer to as operations table ($OT$). The $OT$ has $m \times N$ operational points ($OP$). Each $OP$ is represented as a 4-tuple $< c_k, E_{ik}, t_{ik}, T_{ik} >$. Figure 3.1 shows the $OT$ structure. The values of $t_{ik}$, $E_{ik}$, and $T_{ik}$ can be obtained through profiling, statistical modeling, or a simulation based study. The entries along a column of the $OT$ are arranged in increasing order
of voltage and frequency. This implicitly arranges the entries in a column (from top to down) in the increasing order of $T_{ik}$, and in the decreasing order of $t_{ik}$. We develop a two stage $OC$ selection process to assign voltages and frequencies to different tasks in the queue. The first stage statically assigns voltages and frequencies to each task by comprehensively considering all the $OT$ entries. We develop two complementary polynomial time algorithms, together called the global operational point selection algorithms ($GOPS$), for this purpose. The $OP$ assignments made in this stage satisfy the following constraints in order.

- All task deadlines are met.
- The system steady state temperature never exceeds a threshold limit ($T_{thresh}$).
- All the selected $OP$s are constrained to a temperature window of predetermined size ($T_{window}$).

These $GOPS$ algorithms are global in the sense that they consider the whole set of tasks together at once. Our approach is based on the expected execution time of the tasks rather than the $WCET$s, hence catering for both static and dynamic slack in the system. We monitor the actual execution time values during task execution. Significant deviation of the actual execution time from the expected execution time triggers the second stage of voltage and frequency selection called local operational point selection algorithm ($LOPS$). Thus, $LOPS$ deals with the incremental slack when each single task gets launched for execution.

A set of $OP$s, selected one per task is referred to as an operational chain ($OC$). Thus, an $OC$ dictates the set of voltages and frequencies assigned for the set of tasks in the schedule.
3.2.1 Assumptions

The following assumptions are made in developing the OC selection algorithms.

1. The soft real-time environment guarantees a Quality of Service (QOS) requirement. QOS is defined as the ratio of the number of tasks meeting their deadline to the number of tasks scheduled for execution.

2. The real-time tasks execute long enough so that the chip reaches a steady state temperature (may be different for each task). The transient thermal gradients that arise during task switching affect minimally.

3. The task performance and thermal characteristics are obtained in advance through extensive profiling or an analytical model.

4. The number of voltage and frequency pairs supported by the processor are discrete and finite.

5. Tasks are non pre-emptive and a task schedule constructed by an EDF scheduler is already known.

3.2.2 Procurement of task execution characteristics

In the construction of OT, we need the values $t_{ik}$, $E_{ik}$, and $T_{ik}$ for all the tasks in the task queue. This information will be utilized by our GOPS algorithms to select proper OPs for each task. We obtained these characteristics both using analytical modeling and cycle accurate simulations separately. Since simulations are costly, we use analytical modeling to perform a first order analysis to evaluate the effectiveness of the designed algorithms. Simulations are later used to include the benefits of microarchitectural adaptation since analytical models in this regard are not very accurate.

The analytical model we use is based on a similar model used by [37]. This model is used to calculate $t_{ik}$, $E_{ik}$, and $T_{ik}$ for the tasks operating at different OPs, given the nominal values at the highest (nominal) voltage and frequency settings. The analytical model is presented below for understanding purpose.
For an OP, the normalized voltage $v_{\text{norm}}$ is defined as the ratio between the operating voltage $V$ and the maximum permitted operating voltage $V_{\text{max}}$. Similarly, the normalized frequency $f_{\text{norm}}$ is defined as the ratio between the operating frequency $F$ and the maximum permitted operating frequency $F_{\text{max}}$.

$$v_{\text{norm}} = \frac{V}{V_{\text{max}}}, f_{\text{norm}} = \frac{F}{F_{\text{max}}}$$  \hspace{1cm} (3.1)

The relation between the operating voltage and frequency is approximated as

$$V = aF^{\alpha}$$  \hspace{1cm} (3.2)

where $a$ and $\alpha$ are hardware related parameters. Hence, the relation between $v_{\text{norm}}$ and $f_{\text{norm}}$ can be modeled as

$$v_{\text{norm}} = f_{\text{norm}}^{\alpha}$$  \hspace{1cm} (3.3)

where $\alpha$ is an architecture dependent constant.

Two additional parameters, namely $\rho$ and $\mu$, are introduced to model the dependence of energy consumption, execution time and temperature on the nature of tasks and the underlying architecture. The parameter $\rho$ represents the normalized value of the leakage power consumption to the total power consumption when the processor is operating at $V_{\text{max}}$ and $F_{\text{max}}$. This is a measure of the transistor leakage characteristics, and is highly dependent on the microarchitectural design. The parameter $\mu$ represents the CPU intensity of the task. It is defined as the ratio of CPU computational time to the net execution time for the task. $\mu$ can be obtained by runtime profiling of the tasks. $\rho$ and $\mu$ are given as

$$\rho = \frac{\text{static power}}{(\text{static power} + \text{dynamic power})}$$  \hspace{1cm} (3.4)

$$\mu = \frac{\text{CPU time}}{(\text{CPU time} + \text{memory access time})}$$  \hspace{1cm} (3.5)

The normalized task execution time, $t$ is calculated as

$$t = (1 - \mu) + (\mu)/f_{\text{norm}}$$  \hspace{1cm} (3.6)

This is because the execution time component corresponding to the memory accesses does not change when changing the processor frequency and voltage settings alone. Only the CPU intensive component speeds up (slows down) at a more aggressive (less aggressive) OP.
Both the static and dynamic power scale down as voltage and frequency are decreased. If the static and dynamic powers at the maximum voltage and frequency are represented by \( P_S \) and \( P_D \), then the corresponding values when using \( v_{\text{norm}} \) and \( f_{\text{norm}} \), as given by [27] are

\[
P_{S_{\text{norm}}} = \rho \cdot v_{\text{norm}} \tag{3.7}
\]

\[
P_{D_{\text{norm}}} = (1 - \rho) \cdot v_{\text{norm}}^2 \cdot f_{\text{norm}} \tag{3.8}
\]

Using the definition of static and dynamic power, along with 3.6 yields the value for the normalized energy consumption.

\[
e = (1 - \rho)\mu f_{\text{norm}}^{2\alpha} + \rho(1 - \mu) f_{\text{norm}}^{\alpha} + \rho \mu f_{\text{norm}}^{\alpha - 1} \tag{3.9}
\]

Finally, the normalized steady state temperature is approximately proportional to the power density on the processor chip. Since chip area stays constant, temperature is proportional to net power consumption. Hence, it is modeled as

\[
T = \frac{(1 - \rho)\mu f_{\text{norm}}^{2\alpha + 1} + \rho(1 - \mu) f_{\text{norm}}^{\alpha + 1} + \rho \mu f_{\text{norm}}^{\alpha}}{(1 - \mu) f_{\text{norm}} + \mu) \tag{3.10}
\]

### 3.3 OC Selection

A set of algorithms are developed to select voltages and frequencies for the different tasks scheduled for execution. This selection is based on expected and actual execution times of the tasks. As mentioned earlier, a two-step approach is formulated for the OC selection. The first step, referred to as GOPS, selects an OC based on the expected execution times of different tasks considered. If additional positive or negative slack arises in the schedule during actual execution, it is handled in the second step, referred to as LOPS. Since LOPS algorithm is primarily used to manage and utilize the dynamic slack, it can increase the temperature gradient between the selected adjacent OPs. To minimize this negative effect, the LOPS algorithm selects an alternate OP for a task in the case of positive runtime slack only if this gradient is not significantly aggravated. The next two subsections detail the GOPS and LOPS algorithms.
3.3.1 GOPS Algorithms

The problem of assigning voltages and frequencies to the given task set can be viewed as selection of the proper $OC$ from the $OT$ that satisfies the constraints detailed in Section 3.2. A total of $m^N$ OCs can be constructed from the $OT$ since each task can operate at any of the available $m$ operating modes. Using a brute force approach to compare the merit of all these combinations becomes a computationally daunting task as the number of tasks or operating modes increase. We develop two complementary polynomial time algorithms to select suitable a $OC$ from the $OT$, namely peak reduction and window based selection. The details of these two algorithms are provided in the following subsections.

3.3.1.1 Peak Reduction Algorithm

The Peak reduction algorithm is iterative in nature, and each iteration involves selection of a candidate $OC$ from the feasible pool of $OC$s in the $OT$. The $OP$s for the different tasks are selected such that they fall close to a target temperature set for the iteration. At the end of the iteration, the currently selected $OC$ is compared against the best $OC$ selected so far during the previous iterations. If the newly selected $OC$ is deemed to be more meritorious, it will now be considered as the best $OC$ for future iterations. The following variables are defined and used for the algorithm execution.

1. $Pkr$: A reference to the peak reduction algorithm
2. $o_{chain}$: The $OC$ selected during a $Pkr$ iteration.
3. $CurrentBest\_OC$: A reference to the best $OC$ selected by $Pkr$ until a particular iteration.
4. $target\_T$: The target temperature employed for a $Pkr$ iteration.
5. $feasibilityflag$: A flag indicating whether the chosen $OC$ satisfies the deadline constraints for all the tasks.
6. $doneflag$: A flag signaling algorithm termination.
The algorithm contains two phases $T_{up}$ and $T_{down}$, each of which may execute in multiple iterations. The $T_{down}$ phase is executed first, where the chip temperature profile is iteratively lowered until the algorithm finds a set of OPs that do not satisfy the task deadlines, or the algorithm selects the lowest OPs for all tasks. When the former situation arises, the algorithm switches to $T_{up}$ phase. In this phase, faster OPs are progressively selected leading to higher temperature profile while striving to satisfy task deadlines.

The algorithm initializes by making the OPs corresponding to the highest voltage and frequency a part of $a\_chain$ and $CurrentBest\_OC$.

$T_{down}$ phase In the $T_{down}$ phase, the temperature corresponding to the coolest OP in the $a\_chain$ is selected as the target $T$. In each iteration, OPs for different tasks that are the closest to target $T$ are selected into $a\_chain$. We refer to this selection process as nearestSelect. Once a new $a\_chain$ is completely formed, it is checked for deadline feasibility. The feasibility flag is updated accordingly. If all the deadlines are satisfied, the currently selected $a\_chain$ is compared with $CurrentBest\_OC$ to see how it fares in satisfying the constraints on temperature in the order mentioned in Section 3.2. A conditional update of $CurrentBest\_OC$ occurs accordingly. The algorithm then proceeds to the next iteration and this process repeats until one or few task deadlines are not satisfied. In case the $a\_chain$ selected during an iteration coincides with the $OC$ selected in the previous iteration, the algorithm can get stuck in an infinite loop. To break out of this loop, a new $a\_chain$ is selected which contains the OPs for the different tasks with the next lower voltage and frequency setting compared to the OPs selected in the previous iteration. We refer to this selection process as lowerSelect.

$T_{up}$ phase We reach the $T_{up}$ phase when one or a few task deadlines are not met. Hence, selecting alternate OPs which are slower doesn’t serve the purpose of fulfilling performance constraints. Hence, new $a\_chains$ consisting of more aggressive OPs are selected in this phase. Each iteration in $T_{up}$ phase proceeds as follows. The currently existing coolest OP is taken out of further consideration. The new target temperature is calculated as the temperature of the coolest OP still under consideration. Once target $T$ is calculated, a new $a\_chain$ is selected in
Table 3.1 OC selection schemes for Peak Reduction

<table>
<thead>
<tr>
<th>Scheme</th>
<th>new OC</th>
<th>Use condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowerSelect</td>
<td>$OP_{k(j-1)} \in OC_i \iff OP_{kj} \in OC_{i-1}$</td>
<td>$OC_{i-1} = OC_{i-2}$</td>
</tr>
<tr>
<td>nearSelect</td>
<td>$OP_{kj} \in OC_i \iff (\exists OP_{kl},</td>
<td>T_{kl} - targetT</td>
</tr>
<tr>
<td>alternate</td>
<td>$OP_{kj} \in OC_i \iff (\exists OP_{kl},</td>
<td>T_{kl} - targetT</td>
</tr>
</tbody>
</table>

a fashion similar to nearestSelect. To distinguish the way of selecting targetT from the method used in nearestSelect process, we refer to the current OC selection process as alternateSelect.

The $T_{up}$ phase ends when a deadline feasible OC is selected or there exists no considered OPs in the o_chain that is selected just before $T_{up}$ phase starts.

Listing 3.1 The Peak Reduction Algorithm

```python
Function PeakReduction
    feasibilityflag=1; doneflag=0; initializer();
    while(doneflag==0)
        if(feasibilityflag==0) o_chain=alternateSelect();
        else
            if(o_chain==prev_chain) o_chain=lowerSelect();
            else o_chain=nearestSelect(o_chain.COT);
            feasibilityflag=checkFeasibility(o_chain);
            if(feasibilityflag==1) optimizeChain();
        prev_chain=o_chain;
    return CurrentBest_OC;
End Function

Function alternateSelect
    COPindex=1
    while(COPindex < num_of_tasks)
        o_chain=nearestSelect(o_chain.COT);
        feasibilityflag=checkFeasibility(o_chain);
        if(feasibilityflag==1) optimizeChain();
        COPindex=COPindex + 1 }
    doneflag=1;
End Function

Function optimizeChain
    if(CurrentBest_OC.HOT < Tthresh && o_chain.HOT < Tthresh)
        if(CurrentBest_OC.Sum_Tdiff > o_chain.Sum_Tdiff)
            CurrentBest_OC = o_chain;
    else CurrentBest_OC = o_chain;
End Function
```
The three different OC selection processes are listed in Table 3.1. In the table, the notation OC\textsubscript{i} is used to denote the OC selected at the end of iteration \textit{i}. Listing 3.1 depicts the pseudo code.

**Algorithm runtime**  For a given OT, there can be a maximum of \(m \times N\) \(T_{down}\) iterations and a maximum of \(N - 1\) \(T_{up}\) iterations. The worst case behavior for \(T_{down}\) phase occurs when each iteration leads to change of only one OP in the selected \(o_{chain}\). Similarly, the worst case behavior occurs for the \(T_{up}\) phase when only the last \(o_{chain}\) selected in this phase is deadline feasible. Each \(T_{up}\) or \(T_{down}\) iteration has a time complexity of \(O(mN)\). Hence, the worst case complexity of this algorithm is \(O(m^2N^2)\). Although this seems reasonable, the complexity does not scale well with increasing \(m\). This becomes an issue if a large number of operating modes is considered. An increase in \(N\) can be handled by dividing the task queue into multiple windows and performing OC selection on the different windows separately. In the next subsection, a complimentary algorithm for GOPS that scales better with increasing \(m\) is detailed.

### 3.3.1.2 Window Based Selection Algorithm

An alternative to the peak reduction algorithm is a Window based OP selection (WOPS) algorithm. This algorithm restricts the selection of OPs in favor of reducing the temperature gradients between the tasks. The following variables and metrics are utilized.

1. \(\text{Sum}_{T_{diff}}\): Sum of absolute temperature differences between adjacent tasks in the schedule.

2. \(o_{chain}\): The OC selected during a Pkr iteration.

3. \(\text{CurrentBest\_OC}\): A reference to the best OC selected by WOPS until the specified iteration.

4. Pivot: A selected OP on whose basis other OPs in an \(o_{chain}\) are selected.

5. \(dirn\): A variable specifying the direction (in terms of temperature) in which the next candidate OP for \(o_{chain}\) has to be selected.
The \textit{WOPS} algorithm is iterative in nature. Each iteration employs a pivot \textit{OP} and a virtual temperature window. In each iteration, an \textit{o\_chain} is chosen such that the constituting \textit{OPs} lie close to this window. The pivot for an iteration is simply a candidate \textit{OP} for the first task in the queue. The virtual temperature window constitutes a small temperature range around the pivot’s expected temperature. Since \( m \) such \textit{OPs} are possible, the algorithm executes in \( m \) iterations. Similar to the case of peak reduction, each \textit{o\_chain} selection in \textit{WOPS} is proceeded by a conditional update of \textit{CurrentBest\_OC}. The \textit{WOPS} algorithm also starts by initializing the \textit{CurrentBest\_OC} similar to the Peak reduction approach.

Listing 3.2  The \textit{WOPS} Algorithm

```
1 Function WOPS\_Algorithm
2 initialize();
3 while(o\_chain[0]=selectpivot())!=NULL)
4 { calculateVbounds();
5     while(next\_task\_id < num\_of\_tasks)
6         { o\_chain[next\_task\_id]=selectionAlgo(o\_chain,next\_task\_id,dirn);
7             dirn=setDirection();
8             next\_task\_id++; }
9         current\_schedule\_feasibility=checkFeasibility(o\_chain);
10        if(current\_schedule\_feasibility)
11           { calculateNetSwing(); updateBestChain(); updateFlags(); }
12     }
13 End Function
```

Listing 3.2 shows the pseudo code for our \textit{WOPS} algorithm. The function \textit{selectionAlgo} takes the updated \textit{o\_chain} as the input to find the next \textit{OP} to be included in the chain. A direction variable \textit{dirn} is maintained to guide the selection process. This variable is updated on the fly to force the selection algorithm to choose \textit{OPs} closer to the virtual bounds, if required. This process is illustrated in Figure 3.2. The worst case complexity for one iteration of \textit{selectionAlgo} is \( O(N\log m) \), since there are \( N \) tasks and a binary search can be used to find the candidate \textit{OP} for each task that is closest to the window.

Figure 3.2 shows the \textit{o\_chain} selection round when there are 4 tasks in the system, each of which can operate at 2 different \textit{OPs}. Assume that the point \textit{P11} is selected as the pivot currently. We term the most recently selected \textit{OP} as the \textit{o\_chain} header. The dashed lines in the figure represent the virtual temperature bounds. Initially, the \textit{o\_chain} header lies between the virtual bounds. \textit{dirn} is set to 0 to indicate this. For the second task, there are two potential
OPs available for selection. The selection algorithm selects the closest point to $P_{11}$, which is $P_{21}$. $P_{31}$ is selected for task 3 using similar logic. Since $P_{31}$ falls out of the virtual bounds, it is beneficial for the selection algorithm to select the next OP towards the upper virtual bound, in order to avoid a large thermal gradient with respect to $P_{11}$. $dirn$ is set to -1 to achieve this effect. The selection algorithm thus selects $P_{42}$ instead of $P_{41}$.

Once a complete OC is selected, its deadline feasibility is calculated using the checkFeasibility function, which takes up $O(N)$ processing time. If the feasibility check succeeds, the newly selected OC replaces the CurrentBest_OC utilizing the same logic as that used for Peak reduction. At the end of current iteration, the pivot is marked as invalid and the algorithm starts another iteration by selecting a new pivot.

**Runtime for window based selection** The WOPS algorithm terminates when it runs out of valid pivots to choose from. Since there are $m$ pivot points that can be chosen, the total complexity of the algorithm is $O(Nm\log m)$. The time complexity can be further reduced to $O(N\log m^2)$ by using a binary search for selection of the pivots.
3.3.2 \textit{LOPS} Algorithm

The \textit{LOPS} algorithm takes the best \textit{OC} selected by a \textit{GOPS} algorithm as input and deals with the runtime slack by (potentially) altering each task’s \textit{OP} locally before it starts execution. Since the execution of \textit{LOPS} falls into the actual task schedule, it is designed to be faster. The runtime slack is calculated as the difference of the estimated start time (obtained from schedule constructed by \textit{GOPS}) of a task and the actual start time of the task (obtained during execution). When there is negative runtime slack in the system, the \textit{LOPS} algorithm selects a more aggressive \textit{OP} compared to the one preselected by \textit{GOPS}. On the other hand, when there is a positive runtime slack, the \textit{LOPS} algorithm makes a change to the \textit{OP} selected by the \textit{GOPS} algorithm only if the newly selected point does not cause an additional local \textit{Sum} of \(T_{\text{max}}\) with respect to its adjacent selected \textit{OP}s in the schedule. The \textit{LOPS} algorithm, as listed in Listing 3.3, has a time complexity of \(O(m)\).

\begin{verbatim}
1 Function LOPS_Algorithm
2 int i;
3 if(current_slack == 0) return selected_point;
4 else if(current_slack < 0)
5     for(i=selected_point+1;i<m; i++)
6         if((i.time−selected.time)<current_slack) return i;
7 else if(current_slack > 0)
8     for(i=selected_point−1;i>=0;i−−)
9         if((i.time−selected.time)<current_slack)
10             if(cycleEffect(selected_point,i,task_id)< Twindow/2) return i;
11 End Function
\end{verbatim}

\textbf{Scheduling in periods of persistent slack}  As described earlier, the \textit{GOPS} algorithm creates an initial \textit{OC} based on expected execution times. To account for scenarios leading to continual runtime slack in a single direction, the \textit{GOPS} algorithm creates additional \textit{OC}s based on scaled expected execution times. This helps make comprehensive static decisions that perform better when slack arises. The \textit{OT} is modified by scaling the expected execution times for the different tasks with different scaling factors. For each modified \textit{OT}, an \textit{OC} is chosen using a \textit{GOPS} algorithm. We have limited the bounds of this scaling factor to 0.8 and 1.2 (with a step of 0.05). Using this methodology, we can cater for dynamic slack of 20\% (both
The LOPS algorithm can employ any of these OCs utilizing the different scaling factors during task execution.

The entire task execution schedule is split into windows. Each window contains $W$ tasks. During task execution, a miss counter ($MC$) is employed to monitor the number of deadline misses so far in the schedule. After $i^{th}$ window of tasks finishes execution, we consider the $OC$ selected for a particular scaling factor for deployment in the next window. The chosen scaling factor used for the $i + 1^{th}$ window is adjusted based on the criterion below.

\[
\frac{(MC \text{ value} + W)}{(W \times (i + 1))} < 1 - QOS \quad (3.11)
\]

If the above criterion is satisfied and there exists positive slack in the schedule, the scaling factor employed for the next window is reduced by a step to improve the energy savings. If there exists negative slack in the schedule, the $OC$s corresponding to a step higher scaling factor is utilized by the LOPS algorithm for the next window. The $OC$ selection mechanism also learns how the runtime slack is evolving over time. If a unidirectional slack is continually observed for two successive windows, the step size used for choosing the scaling factor increments by 1. If the slack direction reverses for two adjacent window boundaries, the step size is reset to 1.

### 3.4 Evaluation of the developed DVFS based schemes

We perform both analytical and simulation based analysis to quantify how our GOPS and LOPS lead to increased lifetime expectations. The performance of GOPS algorithms in reducing peak temperatures and thermal gradients on the chip are first evaluated. The runtime for GOPS algorithms in terms of number of iterations is also studied. Such an analysis is essential since GOPS also needs to execute frequently when there is no single set order in which tasks arrive in the task queue. This is followed by an analysis of how the LOPS algorithm caters to the QoS requirements.

For our analysis based on analytical PPR model, a large number of task sets are synthetically generated. Results obtained over 10000 task sets are averaged to smooth out isolated deviant behavior. For the simulation based studies, a simulation framework consisting of Simplescalar simulator, Wattch, and Hotspot is considered. A set of SPEC 2000 benchmarks are used for
analysis. The execution profile of each benchmark is divided into blocks of 10 million instruction each, and configurations are chosen for these intervals. We observed that temperature swings within each block are low (1-4 degrees). The details of our evaluation are presented in the next few subsections.

3.4.1 Experimentation with synthetic task sets

The first set of experiments we performed is intended to test the effectiveness of our $GOPS$ algorithm. We synthetically generated 10000 task sets with $\mu$ values ranging between 0.1 and 1. Task sets of sizes 8, 16, 32, and 64 different periodic tasks are investigated. A $\rho$ value of 0.25 is assumed. This is in par with current predictions in semiconductor industry. Eight different $(V, f)$ combinations are assumed, given by $(V, f) \in \{(1.2, 300), (1.23, 400), (1.35, 500), (1.53, 600), (1.75, 700), (2.0, 800), (2.35, 900), (2.80, 1000)\}$, where $V$ values are in volts and $f$ values are in MHz. These voltage and frequency combinations are taken from a real world processor [105]. The tasks’ steady state temperatures ($SST$s) at the $OP$ with maximum voltage and frequency are selected in the interval of $[310 \text{ K}, 390 \text{ K}]$, linearly increasing with respect to $\mu$. Higher the value of $\mu$, higher is the $SST$ selected. Similarly, the power consumed at the nominal voltage and frequency for the different tasks is selected in the range of $[5 \text{ W}, 10 \text{ W}]$ based on $\mu$. Once the task parameters at the nominal $OP$s are fixed, their corresponding values for the other supported operating modes are calculated using the model detailed in Section 3.2.2. The task execution times at the nominal voltage and frequency are assumed in the range of $[120s, 240s]$. The (percentage of) performance degradation that is accepted to improve reliability is modeled as task stretch factor. The task stretch factors ranging between 5% and 45% are considered in increasing steps of 5%. These stretch factors create static slack in the schedule, that is utilized by our $GOPS$ algorithms for $OP$ selection. The different evaluation parameters employed are listed in Table 3.2. The performance of the different $GOPS$ algorithms is analyzed at these different task stretch factors.

Reduction in thermal gradients Figure 3.3 shows the strength of the $GOPS$ algorithms in minimizing the inter-task $SST$ differences when the number of tasks in the schedule
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>[0.1,1]</td>
</tr>
<tr>
<td>$\rho$</td>
<td>0.25</td>
</tr>
<tr>
<td>$(V,f)$</td>
<td>${(1.2 \text{ v, } 300 \text{ MHz}), (1.23 \text{ v, } 400 \text{ MHz}), (1.35 \text{ v, } 500 \text{ MHz}), (1.53 \text{ v, } 600 \text{ MHz}), (1.75 \text{ v, } 700 \text{ MHz}), (2.0 \text{ v, } 800 \text{ MHz}), (2.35 \text{ v, } 900 \text{ MHz}), (2.80 \text{ v, } 1000 \text{ MHz})}$</td>
</tr>
<tr>
<td>$t_{\text{nom}}$</td>
<td>[120 s, 240 s]</td>
</tr>
<tr>
<td>$P_{\text{nom}}$</td>
<td>[5 W, 10 W]</td>
</tr>
<tr>
<td>$T_{\text{nom}}$</td>
<td>[310 K, 390 K]</td>
</tr>
<tr>
<td>$T_{\text{thresh}}$</td>
<td>353 K</td>
</tr>
<tr>
<td>$T_{\text{window}}$</td>
<td>10 K</td>
</tr>
<tr>
<td>Task stretch factor</td>
<td>5% - 45%, 5% +</td>
</tr>
</tbody>
</table>

Table 3.2  Evaluation parameters used for analyzing effectiveness of GOPS algorithms

considered are 8 (Fig. 3.3 (a)), 16 (Fig. 3.3 (b)), 32 (Fig. 3.3 (c)), or 64 (Fig. 3.3 (d)). The x-axis in the figure indicates the task stretch factors and the y-axis shows the sum of absolute differences between the SSTs of adjacent tasks in the schedule ($\text{Sum}_{\text{tdiff}}$). The max scheme corresponds to operating all tasks at maximum $(V,f)$.

Figure 3.3 shows three trends.

1. $pkr$ does a slightly better job than $WOPS$ is reducing thermal gradients. This trend is expected since $pkr$ chooses the final OPs from a larger pool of candidate OCs when compared to $WOPS$. It is observed that $pkr$ outperforms $WOPS$ when task stretch factor is between 15-35%. Both the algorithms perform similarly in both conditions where very little or very high performance degradation is allowed.

2. As the task stretch factor increases, the GOPS algorithms can do a better job in reducing temperature gradients. This is because the algorithms have at their disposal a larger number of OCs to choose from which satisfy the task deadlines.

3. Both $pkr$ and $WOPS$ algorithms fare well even with a larger number of tasks in schedule. Even though it is tough to find proper OPs that perform closely with respect to their SSTs as the number of tasks increase, this effect is not very pronounced.
Energy savings The energy savings produced by the different GOPS schemes for the task set sizes a) 8, b) 16, c) 32, and d) 64 are shown in figure 3.4. The x-axis in the figure indicates the task stretch factors and the y-axis shows the energy savings as a percentage of the energy consumed using nominal voltages and frequencies.

It is observed that pkr outperforms WOPS in terms of reducing energy when the number of tasks is less. As this number increases, WOPS catches up with and even surpasses pkr. pkr results in very low energy saving when the number of tasks in the schedule is high and the task stretch factor is low. Note that pkr reduces thermal gradients more effectively than WOPS for all task set sizes and stretch factors. Hence, it can be concluded that pkr trades off energy reduction for thermal balance. The energy savings generally decrease slowly as the number of tasks increase. In order to maintain thermal balance, OPs which do not reduce energy the most are selected into the final OC. However, this degradation in energy saving gets less pronounced as higher performance degradation is accepted. It is observed that the GOPS algorithms provide up to about 55% energy savings when the task stretch factor is set at 45%.
Algorithm iterations Though the theoretical maximum number of iterations in Pkr are high \((m \times N + (N-1))\), our experiments revealed that the actual value is much smaller than this bound. For example, the average number of iterations for Pkr is observed to be scaling linearly with the number of tasks. On the other hand, the average number of iterations for WOPS is always bounded by the number of operating modes and is typically less than 4. As the task stretch factor increases, the GOPS algorithms can consider a larger number of candidate OCs which can satisfy all task deadlines. This results in a very slight increase in the average number of iterations. Figure 3.5 shows how the number of algorithmic iterations scale with the performance sacrifice and number of tasks considered. The number of algorithmic iterations for a) 8, b) 16, c) 32, and d) 64 task set sizes are shown along y-axis and the task stretch factor as a percentage value is shown on x-axis.

QoS satisfaction To demonstrate how our algorithms perform with respect to meeting the QOS constraints set, We have scheduled 32 tasks separately for different QOS constraints ranging between 0.91 and 0.99. The peak reduction algorithm is used for GOPS. Each task
window is constrained to the size of 1024 tasks. It has been verified that our LOPS scheme meets the QOS requirements specified. We also observed that as the QOS constraint is tightened, the scheduler utilizes the positive runtime slack more pessimistically, resulting in lower energy savings. However, this difference in savings is marginal (around 5%).

<table>
<thead>
<tr>
<th>QOS Constraint (as %)</th>
<th>QOS Delivered (as %)</th>
<th>Avg. Energy Savings (as %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>92.13</td>
<td>51.75</td>
</tr>
<tr>
<td>93</td>
<td>93.85</td>
<td>50.17</td>
</tr>
<tr>
<td>95</td>
<td>95.54</td>
<td>48.86</td>
</tr>
<tr>
<td>97</td>
<td>97.92</td>
<td>46.47</td>
</tr>
<tr>
<td>99</td>
<td>99.99</td>
<td>46.76</td>
</tr>
</tbody>
</table>

Table 3.3  QOS satisfaction and Energy savings
3.4.2 Simulation based performance reliability tradeoff analysis

Thus far, we have reported the effectiveness of our GOPS and LOPS algorithms in reducing thermal gradients and providing energy savings. The performance, energy, and temperature characteristics for the different tasks are obtained through an analytical model. We now obtain these characteristics through cycle accurate simulations. Since such simulations are costly, we consider only a small task set and provide insight into the reliability improvement provided by our schemes.

3.4.2.1 Simulation framework

For our simulations, we have used the Simplescalar cycle accurate simulator. This simulator can be used to model the execution of applications on a Alpha EV6 like processor. Simplescalar provides different simulation engines with varying degree of detail and accuracy. In particular, we use the sim-outorder engine which considers out-of-order execution of a superscalar processor. sim-outorder is execution driven, making it very accurate for obtaining performance data. Simplescalar is integrated with Wattch, a cycle accurate power modeling tool. Wattch has been integrated with Simplescalar to obtain cycle-by-cycle access characteristics of the different units on the chip floorplan. Wattch is driven by a parameterized power model that estimates the power consumed each cycle based on the aforementioned access characteristics and technology dependent circuit load parameters. To obtain temperature data, we use the
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch, decode, Issue, and commit width</td>
<td>4</td>
</tr>
<tr>
<td>Functional units</td>
<td>4 INT (and FP) ALUs, 1 INT (and FP) MUL/DIV</td>
</tr>
<tr>
<td>L1-D cache</td>
<td>2 KB 4 way</td>
</tr>
<tr>
<td>L1-I cache</td>
<td>2 KB 1 way</td>
</tr>
<tr>
<td>Unified L2 cache</td>
<td>32 KB 4 way</td>
</tr>
<tr>
<td>Technology node</td>
<td>45 nm</td>
</tr>
<tr>
<td>Voltage</td>
<td>1.25 v, 1.15 v, 1.05 v</td>
</tr>
<tr>
<td>Frequency</td>
<td>2536 MHz, 2475 MHz, 2402 MHz</td>
</tr>
</tbody>
</table>

Table 3.4 Simulation parameters used for performance-reliability tradeoff analysis

temperature modeling tool called Hotspot. With the knowledge of the microprocessor floor-plan, Hotspot decomposes the logic circuits into RC networks. Heat sources are modeled as voltage sources to the RC network. The power consumption profiles are provided by Wattch to Hotspot. Figure 3.6 shows the interfaces and data flow between Simplescalar, Wattch, and Hotspot. The floorplan assumed by Wattch are slightly different from the floorplan used by Hotspot. As such, modifications are made to Wattch to produce power profiles amenable to Hotspot. Such modifications were made Prem Kumar Ramesh, who was one of my colleagues in my research group. The baseline parameters used for simulation are shown in Table 3.4.

3.4.2.2 Simulation workloads

A set of 8 SPEC 2000 benchmarks were chosen for experimentation. These benchmarks have been widely used for analysis in the past research. The benchmarks used are listed in Table 3.5 [23]. The inputs for the benchmarks were obtained from Simplescalar website [69].

3.4.2.3 Reliability modeling

Reduction of thermal cycling and chip temperatures on the chip improves its lifetime. To quantify this effect of our schemes, we utilize the chip Mean Time To Failure ($MTTF$), a metric that is widely used to quantify chip reliability [91, 93, 96]. $MTTF$ is defined as the mean expected time to fail of a non-repairable component. Accordingly, the failure mechanisms
investigated should make the chip non-functional and non-repairable. The following failure mechanisms are analyzed: Electromigration, Stress migration, large and small scale Thermal cycles, Time dependent dielectric breakdown, and Negative bias temperature instability. Although there are many other failure mechanisms, we have restricted ourselves to using the most investigated ones, due to availability of near-accurate analytical models to predict the MTTF associated. For each failure mechanism, we calculate the ratio of MTTF when the processor executes tasks with the OPs selected using our scheme to the MTTF obtained when operating at nominal voltage and frequency. In the latter case, the tasks finish faster, and the processor enters a low power mode dictated by the lowest possible voltage and frequency, which is accounted for. Since operating each task at a particular OP results in a different MTTF value, we use weighted harmonic mean to calculate the average MTTF value. The details of the investigated failure mechanisms and our experiments are provided below. Although the temperature profiles on a chip are continuous in nature, we believe that the discrete temperature modeling detailed below gives a good first order estimate of the lifetime improvement.

### 3.4.2.4 Chip Failure Modeling

**Electromigration** The atoms in interconnects are gradually displaced due to momentum transfer by the conducting electrons. Because of this, the atoms get shifted within interconnect and lead to higher resistance values and possibly, shorts. The MTTF due to Electromigration is given by

\[
MTTF_{EM} \propto (J)^{-n}e^{E_{aEM}/KT} \tag{3.12}
\]
where $J$ is the current density in interconnect, $n$ is a material dependent constant, $E_{aEM}$ is the activation energy for electromigration, $K$ is the Boltzmann constant and $T$ is the steady state operating temperature. The value of $J$ is directly proportional to the operating voltage and frequency.

**Stress Migration** Due to differential thermal coefficients in the interconnect material, a thermo-mechanical stress is generated when the interconnect heats leading to migration of atoms. It results in open circuits and high resistance values within interconnects. The $MTTF$ due to Stress migration is given by

$$MTTF_{SM} \propto |T_0 - T|^{-m} e^{E_{aSM}/KT}$$  \hspace{1cm} (3.13)

where $T_0$ is the metal deposition temperature, $T$ is the steady state operating temperature, $m$ is a material dependent constant, $E_{aSM}$ is the activation energy for stress migration and $K$ is the Boltzmann constant.

**Thermal Cycling** We have mentioned both the large and small scale thermal cycling earlier. The $MTTF$ due to large scale thermal cycling is given by

$$MTTF_{LTC} \propto (1/(T - T_{ambient}))^q$$  \hspace{1cm} (3.14)

where $T$ is the steady state operating temperature, $T_{ambient}$ is the ambient temperature and $q$ is the coffin-Manson exponent which is a measure of the effect of thermo-mechanical stress. Small scale thermal cycles cause solder joint failures due to uneven expansion and contraction. We model the $MTTF$ for small scale thermal cycles based on the $MTTF$ for the solder joints. This is given by

$$MTTF_{STC} \propto |T_1 - T_2|^n e^{E_a/KT_{max}}$$  \hspace{1cm} (3.15)

where $T_1$ and $T_2$ are the steady state temperatures of two tasks, $T_{max}$ is the maximum value between $T_1$ and $T_2$, $n$ is the Coffin Manson based exponent, $E_a$ is the activation energy and $K$ is the Boltzmann constant.
### Table 3.6 MTTF model parameters

<table>
<thead>
<tr>
<th>Failure mechanism</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electromigration</td>
<td>$n = 1.1, E_{aEM} = 0.9 eV$</td>
</tr>
<tr>
<td>Stress migration</td>
<td>$m = 2.5, E_{aSM} = 0.9$</td>
</tr>
<tr>
<td>LTC</td>
<td>$q = 2.35$</td>
</tr>
<tr>
<td>STC</td>
<td>$n = -1.9, E_a/K = 1414$</td>
</tr>
<tr>
<td>NBTI</td>
<td>$A = 1.6328, B = 0.07377, C = 0.01, D = 0.06852, \beta = 0.3$</td>
</tr>
</tbody>
</table>

**Negative Bias Temperature Instability** The negative bias applied to the gate results in gradual increase in the threshold voltage and associated decrease in drain current and transconductance. The $MTTF$ for negative bias temperature instability is given by

$$MTTF_{NBTI} \propto \{(\ln(E) - \ln(E - C)) \ast T/e^{-D/KT}\}^{1/\beta}$$

where $A$, $B$, $C$, $D$ and $\beta$ are curve fitting parameters, $K$ is the Boltzmann constant and $T$ is the steady state temperature.

We have used the fitting parameters employed in RAMP model [93] to calculate the $MTTF$ values. The parameters used to calculate $MTTF$ for just the small scale thermal cycles are derived from the model used in [99]. The values for all such parameters are shown in Table 3.6.

#### 3.4.2.5 Performance reliability tradeoff

In this section, the improvement in expected chip lifetime when stretching tasks to fit the acceptable task stretch factor is presented. As the tasks are stretched by decreasing the operating voltage and frequency, chip power consumption goes down. Accordingly, the power density and temperature decrease, leading to longer lifetime expectation. Figure 3.7 shows how the increase in $MTTF$ scales with the task stretch factor.

It can be seen that the $MTTF$ increases increase occurs is steps. This is expected since the available performance and temperature points are discrete in nature. The highest increase is observed corresponding to electromigration. Both peak reduction and window based selection lead to similar $MTTF$ improvements at significant task stretch factors. Since only a small set of
voltages and frequencies are available, the improvement in $MTTF$ also tapers off with increase in task stretch factor. As the performance traded off for reliability slowly increases, we observe that window based selection exploits the performance sacrifice first. Peak reduction does not result in any reliability improvement until the task stretch factor increases over 5%. This is a consequence of the inherent differences in the way these two schemes select $OP$s. In particular, peak reduction uses the lowest temperature point in each iteration as the target temperature for next iteration. If the newly selected chain does not satisfy performance constraint, the algorithm reverts back to selecting the nominal $OP$s. Window based selection always considers a candidate $OP$ for the first task for deciding the target temperature.

It should be noted a larger increase in lifetime expectation is hindered by the course-grained performance-temperature points provided by the different voltage and frequency settings. To further improve the $MTTF$ value, more operating points as well as even lower power operating points are needed. There are two architectural alternative available in this regard. Firstly, more voltage and frequency settings can be provided. Secondly, the aggressiveness of a few architectural components can be adjusted (microarchitectural adaptation). In this dissertation, both $DVFS$ and microarchitectural adaptation are considered together. The motivation behind this synergistic strategy is explained in the next subsection.
### Table 3.7 Operating voltages and frequencies for Intel Pentium M processor

<table>
<thead>
<tr>
<th>Mode id</th>
<th>Voltage (Volts)</th>
<th>Frequency (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.484</td>
<td>1.6</td>
</tr>
<tr>
<td>2</td>
<td>1.420</td>
<td>1.4</td>
</tr>
<tr>
<td>3</td>
<td>1.276</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>1.164</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>1.036</td>
<td>0.8</td>
</tr>
<tr>
<td>6</td>
<td>0.956</td>
<td>0.6</td>
</tr>
</tbody>
</table>

#### 3.5 Performance reliability tradeoff using DVFS and microarchitectural adaptation

#### 3.5.1 Need for considering DVFS and microarchitectural adaptation together

Consider the results we obtained through interval simulations shown in Figures 3.8 and 3.9, which depict the impact of (a) L1 instruction cache (IL1) associativity (assoc) and (b) operating voltage and frequency (together referred to as VF) on the normalized performance (Figure 3.8) and power consumption (Figure 3.9) for the SPEC 2006 benchmarks astar, xalancbmk, tonto, and milc executing on an adaptive Intel Nehalem processor [49]. The execution is simulated using Sniper simulation platform [15] and the cache size per each associative way is 4 KB. More details on the simulation platform are provided in Chapter 4. The adaptive processor is assumed to support three levels of cache associativity- 2, 4, and 8. Similarly, the processor supports 6 different VF pairs, shown in Table 3.7. The values listed in the table are taken from the datasheet for an Intel M processor [48] based on Nehalem microarchitecture. From the data obtained through simulations, we observe that decreasing the IL1 assoc from 8 to 2 does not affect normalized performance significantly for astar and milc. Simultaneously, a significant impact is noticed for tonto and xalancbmk. It must be noted that the core voltage and frequency are significant factors driving performance for all of these benchmarks. If a 15% reduction in power is desired, assoc can be lowered to 2 for astar and milc, conserving 15.5% power in both cases. If voltage and frequency are scaled instead of IL1 adaption, it results in higher performance loss to obtain this power reduction (25% in both cases). In case of xalancbmk and tonto, trading off performance for power reduction by adapting IL1 assoc...
results in a larger performance loss (32% and 12.5% respectively) when compared to operating in mode 2 to satisfy power constraint (21% and 9% performance loss respectively). From the above discussion, it is clear that the effectiveness of trading off performance for power consumption, and consequentially reliability, through \textit{DVFS} or microarchitectural adaptation is application dependent. Hence, a unified scheme that considers both \textit{DVFS} and microarchitectural adaptation together can lead to a better tradeoff.

### 3.5.2 Selection of adaptive microarchitectural components

A large number of microarchitectural components can be adapted [31]. In this dissertation, a small subset of those components are used for performance reliability adaptation. Essentially, the components contributing largely to overall chip power consumption are chosen. Figure
Figure 3.10  Power consumption breakdown among different units on Alpha EV6 floorplan

Table 3.8  Adaptive hardware configurations

<table>
<thead>
<tr>
<th>Component</th>
<th>Considered configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 D-cache associativity</td>
<td>{1, 2, 4}</td>
</tr>
<tr>
<td>Int Exec.</td>
<td>{1, 2, 4}</td>
</tr>
<tr>
<td>FP Add</td>
<td>{1, 2, 4}</td>
</tr>
<tr>
<td>$(V_{involts}, f_{inMHz})$</td>
<td>{$(1.25, 2536)$, $(1.15, 2475)$, $(1.05, 2420)$}</td>
</tr>
</tbody>
</table>

3.10 shows the relative power consumption of different units on the floorplan of an Alpha EV6 processor when executing SPEC 2000 benchmarks. In the figure, the x-axis denotes the different benchmarks considered and y-axis denotes the power consumption in watts. From the figure, it is clear that the L1 data cache, Integer ALU, and the FP ADD unit are the three most power hungry components. As such, these components are chosen for adaptation along with DVFS. The configurations that are considered for adaptation are listed in Table 3.8. Also, different components affect the performance-power balance differently; hence, it is not guaranteed that higher performance levels necessarily translate to higher chip temperatures. This insight is used to remove inefficient OPs on a per-application basis that result in lower performance and higher temperature when compared to another valid configuration for the same application. Details on this pruning strategy is made clear in Chapter 4.
3.5.3 Performance reliability tradeoff considering both DVFS and microarchitectural adaptation

Figure 3.11 shows how a combination of DVFS and microarchitectural adaptation can further lead to improved lifetime expectations when WOPS algorithm is utilized. In the figure, the x-axis denotes the task stretch factors considered and y-axis denotes the MTTF values. The values along y-axis are a ratio of MTTF improvements observed for the cases of DVFS plus microarchitectural adaptation and DVFS alone.

The MTTF improvement in this case is the highest for short term thermal cycling behavior. Since a large number of temperature points are available for each task to choose from, it becomes easier to reduce thermal gradients as well. It is also observed that the increase in MTTF ratio does not monotonically scale with performance degradation. For small task stretch factors, window based selection even results in lower MTTF when compared to utilizing just DVFS. Some OPs that are utilized in the latter case are now eliminated due to their inferiority. As window based selection tries to restrict all OPs within a window, unavailability of a few such configurations leads to choosing of higher temperature OPs. It is also observed that the actual MTTF increases as the task stretch factor increases.
3.6 Conclusion

In this chapter, the issue of microprocessor performance reliability tradeoff is dealt with. A real-time task execution environment is considered. A two stage methodology for selecting good hardware operating modes to improve reliability in periods of available slack in the task schedule is developed. This methodology leverages on the knowledge of hardware-software interaction characteristics which have been obtained separately through analytical modeling and cycle accurate simulations. Both DVFS and microarchitectural adaptation are utilized to provide different operating modes on a microprocessor. The results obtained through experimentation indicate that the developed schemes do a very good job in decreasing chip temperatures and thermal gradients. The thermal gradients for a task schedule with 32 tasks are reduced by as much as 80% when the performance degradation accepted is 45%. A combination of DVFS and microarchitectural adaptation led to an 2.5-15 fold increase in expected chip MTTF values corresponding to different failure mechanisms and 41 fold in chip MTTF values corresponding to short term thermal cycling, when 10% performance degradation is allowed.
CHAPTER 4. ADAPTIVE MICROARCHITECTURAL CONFIGURATION SPACE PRUNING

This chapter details the methodology developed for adaptive configuration space pruning. The adaptive configuration space is introduced formally and a three stage approach designed for pruning is detailed. The effectiveness of these schemes in retaining the most relevant/beneficial configurations is analyzed. This configuration space reduction enables the further development of a static cum dynamic framework for performance-power tradeoff based on profiling/simulation data. The actual methodology for performance-power tradeoff follows in Chapter 5.

4.1 Introduction

Let there be $K$ adaptive hardware components or control knobs ($CK$). Let the $i^{th}$ component be configurable in $w_i$ ways, represented by $C_i = \{c_{i1}, c_{i2}, ..., c_{iw_i}\}$, where $C_i$ is totally ordered under $<$. Assuming that the configuration choices for different components are independent, the total number of possible architectural configurations is given by $T = \prod_{i=1}^{K} w_i$. The set of all possible configurations makes the configuration set/space ($S$). The $j^{th}$ configuration in $S$ is represented as

$$s_j = \{c_{1a_1}, c_{2a_2}, ..., c_{Ka_K}\} \mid \forall \ 1 \leq a_i \leq w_i, \ c_{ia_i} \in C_i$$

(4.1)

For an application, let the normalized performance delivered ($P_{\text{norm}}$) and the normalized average power consumed ($W_{\text{norm}}$) by configuration $s_j$ be denoted by $P_j$ and $W_j$, respectively. Performance and power values are normalized with respect to the corresponding values obtained with the maximal or non-adapted configuration.

The large value of $T$ is a major hindrance to design lightweight microarchitectural adaptation methods. The problem is identified in previous research (e.g., [31, 65]). For example, the
authors in [31] identify 627 billion configurations arising from adapting 14 different hardware components simultaneously. To counter complexity, peephole optimizations are applied to windows of instructions. The solutions obtained are impressive, yet suboptimal when compared to an ideal scheme that considers the entire application execution profile holistically. A similar condition is also encountered by authors in [61], where the effects of adapting the different components are considered individually, rather than in a holistic manner. In order to enable the development of an adaptation scheme that overcomes these limitations, reduction of $T$ is necessary. Reduction of $T$ also motivates the design of adaptive microarchitectures.

To make $T$ small, reduction in $K$ and all $|w_i|$s is required. Since the initially considered configuration space is very large, a comprehensive analysis of such a space is expensive. However, a detailed analysis leads to the selection of the most beneficial configurations. We develop a three step pruning methodology to achieve the configuration space pruning. Following these steps, the configuration space considered reduces progressively while the analysis complexity increases. This enables us to reap the benefits of comprehensive analysis within a short course of time. Each of these steps can be viewed as a transformation of $S$ to $S'$ using a specific criterion, such that $|S'| \leq |S|$. Further, reducing $K$ also reduces the hardware complexity in provisioning the associated controls. The three steps are as follows. The details of these steps are presented in the following sections.

1. Selection of advantageous control knobs ($SACK$): This step targets the reduction of $k$ and all $w_i$s for a given microarchitecture. The recommendations following this step can be used by chip designers to provide the appropriate adaptive controls.

2. Elimination of inferior configurations ($ELIC$): This step eliminates configurations that do not perform well but consume unjust power when compared to any other configuration in $|S|$. This step chooses the most beneficial configurations to use for adaptation while executing a particular application.

3. Configuration set selection for runtime ($CSSR$): This step brings down $T$ to any desired number. The target size of $|S'|$ can be decided on the basis of expected adaptation granularity and complexity.
4.2 Selection of Advantageous Control Knobs (SACK)

Let $S_{\text{max}}$ (or simply $\text{max}$) represent the maximal processor configured in terms of consuming the highest amount of power $W_{\text{max}}$, while most probably delivering the highest performance $P_{\text{max}}$. Similarly, let $S_{\text{min}_j}$ represent the processor configuration which exactly matches $S_{\text{max}}$ with the exception of configuring the adaptive component $j$ minimally. Let this configuration deliver the performance $P_{\text{min}_j}$ while consuming power $W_{\text{min}_j}$. In this step, an adaptive component is eliminated if its adaptation cannot tradeoff at least 10% of overall power consumption. This value is chosen so as to avoid unnecessary control and hardware overhead associated with adapting the component while reaping insignificant benefits. The pruning criterion can be represented as

\[
\{ s_i \notin S' \land C_{jw_i} \notin s_i \} \iff t_{p_j} < 0.1
\]

where $t_{p_j}$ is the tradeoff potential for the adaptive component $j$, given by

\[
t_{p_j} = W_{\text{max}} - W_{\text{min}_j}
\]

We initially consider a processor with the adaptive units and available adaptive levels shown in Table 4.1. The adaptive configuration space contains 1,728 configurations. The maximum adaptive level mentioned for each component is based on an Intel Nehalem family processor [49]. The minimum levels are limited by technology for all controls except for instruction window size, where performance implications led to the limit. For the case of caches, the total cache size scales up proportionately with the number of associativity levels. These components selected have been adapted in previous research and the hardware complexity involved is shown to be acceptable ([13, 55, 82]).

We first observed that reducing cache associativity from 2 to 1 does not result in significant reduction in power, but has an adverse effect on performance. Hence, we restrict the minimum associativity to 2. This leads to $T = 729$ configurations.

We explored the merit of adapting the six components individually by simulating benchmarks from the SPEC 2006 suite [39] on an Intel Nehalem processor using the Sniper simulation platform [15]. All the simulation analysis performed from this point is based on observations
<table>
<thead>
<tr>
<th>id</th>
<th>CK name</th>
<th>Adaptive configuration levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dispatch width (DW)</td>
<td>4, 2, 1</td>
</tr>
<tr>
<td>2</td>
<td>Instruction window size (IW)</td>
<td>128, 64, 32</td>
</tr>
<tr>
<td>3</td>
<td>L1 Instruction cache associativity (IL1)</td>
<td>8, 4, 2, 1</td>
</tr>
<tr>
<td>4</td>
<td>L1 Data cache associativity (DL1)</td>
<td>8, 4, 2, 1</td>
</tr>
<tr>
<td>5</td>
<td>L2 cache associativity (L2)</td>
<td>8, 4, 2, 1</td>
</tr>
<tr>
<td>6</td>
<td>(Voltage (V), Frequency (GHz)) (VF)</td>
<td>(1.484, 1.6), (1.228, 1.2), (1.036, 0.8)</td>
</tr>
</tbody>
</table>

Table 4.1 Considered adaptive components and adaptations

that we made for these benchmarks. *Sniper* is a high-speed and accurate x86 simulator that employs an accurate mechanistic analytical model which drives the timing simulation of an individual core. A branch predictor, memory hierarchy, cache coherence and interconnection network simulators built into *Sniper* determine miss events. The analytical model derives the timing for each interval between successive miss events. The cooperation between the analytical model and the miss event simulators enables the accurate modeling of process execution. The simulator is also integrated with McPAT [68] to produce accurate power estimations.

4.2.1 Observations

For each benchmark, execution under seven configurations represented by $\{S_{\text{max}}, S_{\text{min}}_1 - S_{\text{min}}_6\}$ are simulated. The resulting performance and power consumption values are recorded and normalized with respect to the corresponding values obtained under $S_{\text{max}}$. Table 4.2 shows the maximum tradeoff potential ($tp$) values for the different components considered for adaptation. We observed that voltage and frequency scaling control followed by dispatch width have the highest tradeoff potential. Further, the components $DL1$ and $L2$ fit the pruning criterion mentioned in Equation 4.2, and are thus eliminated. This reduces $K$ by two and brings down $T$ to $3^4 = 81$ configurations. This is a significant reduction. Altering $IW$ provides very fine variations in performance and power. This component is retained along with $DW$, $L1I$, and $VF$ to provide fine-grained control.

To quantitatively observe the tradeoff potential for the entire chosen configuration space, we tabulate (Table 4.3) the average, minimum, and maximum variations in performance and
<table>
<thead>
<tr>
<th>id</th>
<th>CK name</th>
<th>tp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dispatch width (DW)</td>
<td>0.31</td>
</tr>
<tr>
<td>2</td>
<td>Instruction window size (IW)</td>
<td>0.12</td>
</tr>
<tr>
<td>3</td>
<td>L1 Instruction cache associativity (IL1)</td>
<td>0.22</td>
</tr>
<tr>
<td>4</td>
<td>L1 Data cache associativity (DL1)</td>
<td>0.03</td>
</tr>
<tr>
<td>5</td>
<td>L2 cache associativity (L2)</td>
<td>0.04</td>
</tr>
<tr>
<td>6</td>
<td>(Voltage (V), Frequency (GHz)) (VF)</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 4.2  $tp$ for the considered adaptive components

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Measured quantity</th>
<th>variation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Minimum</td>
<td>47.9</td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td>80.3</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>71.7</td>
</tr>
<tr>
<td>Power</td>
<td>Minimum</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td>73.9</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>69.9</td>
</tr>
</tbody>
</table>

Table 4.3  Performance-power variations provided using the chosen configuration space

power consumption that can be provided for the studied set of benchmarks. We observe that the envisioned configuration space is sufficient to cater to varied user demands.

Provision of adaptive controls in hardware also leads to power consumption overhead. It also leads to a larger number of failure points in the system. $SACK$ can reduce the ill effects concerning the above two scenarios. It is true that elimination of certain adaptive controls can lead to suboptimal tradeoff decisions. However, the resulting simplicity makes a case for the implementation of such hardware controls. Further, the reduced configuration space can be analyzed accurately in more detail to counter the sub-optimality of tradeoff. Finally, the $SACK$ step considers adaptive controls closest to the core presently. The analysis performed can be extended to larger configuration spaces including off-chip adaptive controls as well. However, such an analysis is not a part of the current research.
4.3 Elimination of ineffective configurations (ELIC)

Reduction in configuration space in this step is based on a per-application analysis. We make the following observation to further reduce the configuration space. Adaptation of different control knobs impacts performance-power balance differently. As a consequence, it is not guaranteed that $P_i > P_j$ whenever $W_i > W_j$ for two configurations $s_i$ and $s_j$. If that is the case, $s_i$ can be removed from $S$. Using this criterion, all ineffective hardware configurations are eliminated. Since the obtained performance and power values for the various configurations are dependent on the application under consideration, this pruning step is independently performed for each benchmark separately. The pruning criterion can be represented as

$$s_i \notin S', \exists s_j \in S \mid \{P_i < P_j\} \land \{W_i > W_j\}$$

(4.4)

Pruning is done by sorting configurations in decreasing order of $P$ values and then inspecting the $W$ values on the sorted list. If a configuration consumes more power than its predecessor, it is removed. For $n$ configurations, the sorting and the inspection processes take $O(n \log n)$ and $O(n)$ time respectively.

4.3.1 Observations

We found that the application of ELIC step reduced the configuration space significantly. The average number of configurations retained is just 24, while the actual number is different for different benchmarks. The most number of configurations were retained for libquantum (38), while the least number were retained for astar and gobmk (17 each). The actual number of retained configurations for the different benchmarks considered is shown in Figure 4.1. In the figure, the X-axis shows the different benchmarks and the Y-axis shows the number of configurations eliminated by this pruning step. One important observation we make is that although an individual benchmark can benefit only from a few configurations, different applications benefit from different subsets of configurations. Overall, we noticed that 72 out of the 81 configurations considered are useful for at least one application. An important observation we make is that at high $(V,f)$ or at intermediate $(V,f)$, a low dispatch does not deliver good
Figure 4.1 Number of configurations eliminated by ELIC

performance for the power consumed, and the corresponding combinations have either been eliminated or sparingly retained for all benchmarks.

Further, each knob does require all the adaptive levels considered, and the distribution of their usage is fairly uniform. Figure 4.2 shows the percentage utilization of different adaptive configuration levels for each considered adaptive control in the configurations retained after ELIC step. This figure shows the frequency in percentage (Y-axis) with which each adaptive configuration for every adaptive component (X-axis) is used in beneficial configurations for the considered benchmarks. It is observed that the lowest adaptation level is retained with a higher frequency for the different components other than dispatch width. This can be explained by a stronger positive correlation between normalized power and performance delivered at lower power levels, than at intermediate or higher power levels. If one adaptive configuration has to be removed further, the lowest dispatch width can be targeted for elimination.

4.4 Configuration Set Selection for Runtime (CSSR)

The previous two steps eliminate several less effective CKs and configurations. The goal in this step is to reduce $|S'|$ to a target number $k$. For our experiments, we have set the value of $k$ at 16. Thus far, the configurations eliminated are deemed ineffective. Further pruning
Figure 4.2 Usage frequency of the individual adaptive settings for the considered adaptive components

<table>
<thead>
<tr>
<th>Pruning method</th>
<th>pruning criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merit based selection</td>
<td>Retain a set of configurations that provide the best performance per watt</td>
</tr>
<tr>
<td>Bound based selection</td>
<td>Retain a set of configurations that are spread evenly along the possible performance-power spectrum</td>
</tr>
<tr>
<td>Neighborhood based selection</td>
<td>Aggressively prune subsets of configurations that behave similarly</td>
</tr>
</tbody>
</table>

Table 4.4 Different pruning methods for CSSR

would potentially eliminate good configurations to reduce runtime adaptation overhead. Hence, the criterion used for further pruning has to be sensitive to the tradeoff effectiveness. Three different pruning criteria are individually considered and the pruning has been implemented accordingly. The three different pruning methods and criteria are listed in Table 4.4. The details of these methods follow in the next three subsections.

4.4.1 Merit based selection

The merit based selection pruning methodology selects the $k$ best configurations out of the configuration space when compared with each other using a metric merit ($M$).
The merit of a particular configuration is defined by the ratio of how much performance advantage the configuration provides with respect to the configuration with the lowest performance point \( P_l \) to the additional power it consumes with respect to the configuration with the lowest power consumption \( W_l \). The \( M \) value for the configuration with \( P_l \) and \( W_l \) is set to 1. For each other configuration \( i \), its merit value \( M_i \) is calculated as

\[
M_i = \frac{P_i - P_l}{W_i - W_l}
\]  

(4.5)

Merit-based CSSR performs the following actions, in order.

1. The merit \( M \) for each configuration is calculated. This step incurs \( O(n) \) time complexity.

2. Configurations having the \( k \) highest \( M \) values are retained in the configuration space. An approach similar to heap sort is utilized for selection of configurations with high \( M \) values. This step incurs a time complexity of \( O(k\log n) \).

The algorithm inserts all the configurations into a top-down heap with the \( M \) value as the key. This ensures that the configuration with the highest merit value is at the root position. Merit-based CSSR then iteratively removes the root of the heap, selects the corresponding configuration, and re-heaps until all the required \( k \) values are chosen. The number of such iterations are thus \( k \) and the complexity of each iteration is \( O(\log n) \). The algorithm uses \( O(n) \) extra space for the heap. While merit-based CSSR guarantees the selection of \( k \) configurations, it does not guarantee uniform sampling over the possible range of performance and power values, which might lead to higher \( TE \) when satisfying user demands. The pseudo code for merit-based selection for CSSR is shown in Algorithm 1.

If all the \( P_i \)s and \( W_i \)s are plotted along two parallel straight lines, and the points on the lines that correspond to the same configuration \( s_i \) are joined, the selection process can be graphically viewed as selecting the configurations whose joining lines have the lowest \( k \) slope values. This situation is shown in Figure 4.3, where \( n = 8 \), \( k = 4 \). In the figure, the performance and power consumption characteristics of an initial set of configurations are plotted along two parallel lines. The configurations are represented as \( s_i \), where \( i \) varies between 1 and 8. The merit values for all the configurations are calculated as per Equation 4.5. Then, the configurations with the four largest merit values are retained in the configuration space.
Algorithm 1 Merit based selection

for $s = s_1 \rightarrow s_n$ do
    $Merit \leftarrow CalcMerit(s)$
    HeapInsert(heap, $s$, $Merit$)
end for

for $i = 1 \rightarrow k$ do
    $s \leftarrow RetrieveHead(heap)$
    Select($s$)
    ReHeap(heap)
end for

4.4.2 Bound based selection

Since user demands can be diverse in nature, it makes sense to retain configurations spread out uniformly along the performance-power spectrum to be able to satisfy diverse demands well. The provided performance spectrum is divided into intervals and configurations that provide performance values closest to the interval bounds are chosen. Bound based selection for CSSR performs the following actions in the scenario where $k$ out of $n$ configurations have to be retained.

1. With the knowledge of the highest ($P_h$) and lowest ($W_h$) provided performance levels, the configuration space is divided into $k-1$ equal sized intervals such that the first interval starts at $P_l$ and the last interval ends at $P_h$. The intervals are thus separated by bounds $b_j$ where $1 \leq j \leq k$.

2. Bound-based CSSR selects one configuration that is closest to each bound and retains it in the configuration space. Since each configuration $s_i$ has associated $P_i$, a Distance metric ($DM$) is used to quantify the distance between configuration $s_i$ and a bound $b_j$. 

\[ S = \{ S_1 (0.5,0.45), S_2 (0.55,0.5), S_3 (0.65,0.6), S_4 (0.7,0.65), S_5 (0.75,0.7), S_6 (0.8,0.85), S_7 (0.9,0.9), S_8 (1,1) \} \]
\[ M = \{ 1,1,1,1,1,0.75,0.88, 0.9 \} \]
\[ S' = \{ S_1 (0.5,0.45), S_2 (0.55,0.5), S_3 (0.65,0.6), S_4 (0.7,0.65) \} \]
We define $DM_j$ for a configuration $s_i$ and a bound $b_j$ as

$$DM_j = |P_i - b_j|$$

(4.6)

3. For each $b_j$, the configuration providing the least value for $DM_j$ is retained into the final configuration space. We designate this configuration as the minimizer for $DM_j$ ($i = mzr_{DM_j}$). This ensures the uniform sampling of the configuration space. Ties between two equally close configurations can be broken in favor of the one that provides a higher $P_{\text{norm}} \cdot W_{\text{norm}}$.

Although it appears that the time complexity of the algorithm is $O(kn)$ (we have to calculate $DM$ for every configuration with respect to every bound), it can be computed in $O(n)$ time by observing the following. \(\{i = mzr_{DM_i} \land j = mzr_{DM_m}\} \land \{b_i \geq b_m\} \Rightarrow \)

$$\{P_i \geq P_j\} \land \{W_i \geq W_j\}$$

(4.7)

Notice that this condition holds since configurations that do not satisfy this property are removed from the configuration space by the $ELIC$ step. Since the same configuration can be selected for multiple bounds, an additional constraint of selecting a previously unselected configuration into the configuration space for every bound is enforced.

The pseudo code for the algorithm is shown in Algorithm 2. An example demonstrating the working of this algorithm is shown in Figure 4.4, where $n = 8$, $k = 5$. In the figure, the performance and power consumption characteristics of an initial set of configurations are plotted along two parallel lines. The configurations are represented as $s_i$, where $i$ varies between 1 and 8. The dashed vertical lines represent the bounds generated. $UM$ is initially set to a large value, say 1. The algorithm starts by considering configuration $s_1$ for $b_1$. The corresponding $UM_{-1}$ is measured as $|0.5 - 0.5| = 0$. As $UM$ decreased since the previous case (1), the next configuration is now considered for the same bound. $UM_{-1}$ for $s_2$ is calculated as 0.05. Since we observe an increase in $UM$, $s_1$ is selected as the minimizer for $b_1$. The algorithm then considers $s_2$ for $b_2$, and the process continues.
Algorithm 2 Bound based selection

\[
b \leftarrow b_1
\]
\[
U_{Best} \leftarrow U_{MAX}
\]
\[
U_{Best}\ Conf \leftarrow NULL
\]
\[
s \leftarrow s_1
\]

while \( s \neq NULL \) do

if \( b = NULL \) then

break

end if

if \( UCalc(s, b) < U_{Best} \) then

\[
U_{Best} \leftarrow UCalc(s, b)
\]

\[
U_{Best}\ Conf \leftarrow s
\]

\[
s_{prev} \leftarrow s
\]

\[
s \leftarrow (s \rightarrow next)
\]

else

Select\( (s_{prev})\)

\[
b \leftarrow b_{next}
\]

\[
U_{Best} \leftarrow U_{MAX}
\]

\[
U_{Best}\ Conf \leftarrow NULL
\]

end if

end while
4.4.3 Neighborhood based selection

This pruning method leverages on the fashion in which the different configurations are distributed in the performance-power space. This method also eliminates more configurations than the above methods and still meets the user demands with a defined degree of precision. Let us first consider an example to motivate neighborhood based selection approach. Figure 4.5 shows the configurations retained for the soplex benchmark after the ELIC step as a graph. The configurations retained in the ELIC step are shown as nodes in the graph. The nodes whose performance and power values are within 5% range of each other are connected using an edge. Configurations connected by an edge are referred to as neighbors.

It can be seen that the graph contains a number of connected components. Since connected nodes deliver similar performances while consuming similar amount of power, only one node from each pair of connected nodes needs to be kept in the final configuration set. This guarantees that the retained configuration space can cater to user demands almost as well as the un-pruned set. For example, in the figure, the connected components are: \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6, 7\}, \{8, 9\}, \{10, 11\}, \{12, 13\}, \{12, 14\}, \{13, 14\}, \{15, 16\}, \{17, 18\}. In this example, if configuration 13 is retained in the final configuration space, then configurations 12 and 14 can be eliminated. If any of the eliminated configurations was optimal to satisfy a user demand, then its removal increases the inaccuracy in satisfying any user demand by a maximum of 0.05. Following this discussion, if a total of only 11 configurations can be retained in the final configuration space, a set of good configurations chosen would be \{1, 2, 3, 4, 5, 6, 8, 10, 13, 15, 17\}.

Figure 4.5  Graphical representation of the retained configuration space for soplex benchmark after ELIC pruning step
### 4.4.3.1 Satisfying a user demand

An individual user demand \((U_d)\) is represented as a 2-tuple \(U_d = < P_d, W_d >\), specifying the normalized performance \((P_d)\) and power consumption \((W_d)\). We use the Inaccuracy \((I)\) in satisfaction of \(U_d\) as a measure of imprecision in demand satisfaction. For a given configuration set \(S\) and a user demand \(U_d\), \(I\) is given by

\[
I(U_d, S) = \min_{s_i \in S} \max(\max(P_d - P_{s_i}, 0), \max(W_{s_i} - W_d, 0))
\]

The goal is to transform \(S\) to \(S'\) such that for any arbitrary \(U_d\), \(I(U_d, S') - I(U_d, S) < p \mid |S'| \leq k\), where \(p\) is the acceptable loss in precision.

We use a greedy method, which we call neighborhood-based elimination, to eliminate a subset of configurations. We represent the configurations in \(S\) as a graph \(G(V, L)\), where \(V\) represents the set of vertices and \(L\) represents the set of links. The construction of \(G\) follows these properties.

- Each configuration \(s_i \in S\) is represented by a vertex \(v_i \in V\).
- Vertices representing two configurations \(s_i\) and \(s_j\) in \(S\) are connected by an edge in \(G\) if \(\max(|P_{s_i} - P_{s_j}|, |W_{s_i} - W_{s_j}|) < p\). In this case, \((v_i, v_j) \in L\). This is referred to as the closeness property. For our experiments, we chose \(p\) to be 0.05. The condition guarantees that utilizing \(s_i\) to satisfy any arbitrary \(U_d\) instead of \(s_j\) guarantees that \(I(U_d, S') - I(U_d, S) < p\).

For each vertex in \(G\), a neighbor set is constructed. The neighbor set for a given vertex is defined as the set of vertices that have a link to it. The construction of all neighbor sets can be collectively represented as \(\forall v_i, v_j \in V, v_j \in N(v_i) \Leftrightarrow (v_i, v_j) \in L\), which incurs \(O(n^2)\) time if \(|S| = n\). The pruning problem translates to selecting a set of \(k\) vertices (denoted by the set \(V'\)) from \(V\) such that \(V'\) includes at least one vertex from each neighbor set. This follows from the argument that utilization of a configuration corresponding to any one vertex in the neighbor list for a vertex \(v_i\) will satisfy any arbitrary user demand that is originally satisfied by the configuration corresponding to \(v_i\) without increasing \(E\) by more than \(p\). Since the neighbor
sets exhibit symmetric property i.e., $v_j \in N(v_i) \Leftrightarrow v_i \in N(v_j)$, the pruning problem can also be translated to selection of $V'$ vertices from $V$ such that $|V'| \leq k$ and $\bigcup_{v_i \in V'} N(v_i) = V$.

In addition to the neighbor sets, the neighborhood-based pruning algorithm also maintains a satisfaction set ($T$) which contains the configurations for whom the closeness property is satisfied due to configurations in $S'$. Both $T$ and $S'$ are initialized to a null value.

The pruning algorithm executes iteratively to select configurations into $S'$. In each iteration, the following actions are performed in order.

1. A configuration whose representative vertex in $G$ has the highest cardinality of neighbor set is selected. Let this configuration be represented by $s_m$ and the corresponding vertex in $G$ be represented by $v_m$.

2. $s_m$ is included into $S'$ i.e., $S' = \{s_m\} \cup S'$.

3. All the vertices in $N(v_m)$ are included into $T$, i.e., $\forall v_i \in N(v_m)$, $T = \{v_i\} \cup T$. If the vertex being included already exists in $T$, it is not added again.

4. All neighbor sets are updated to remove vertices added to $T$ in the present iteration.

Each algorithmic iteration requires $O(n^2)$ time. The algorithm terminates when either of the following situations arise.

1. The cardinality of $S'$ reaches the target, i.e., $|S'| = k$.

2. The closeness condition is satisfied for all, i.e., $|T| = |S|$.

The worst case runtime for the algorithm is $O(kn^2)$. In case the algorithm terminates due to condition 1 above, but $|T| \neq |S|$, it implies that the algorithm failed to prune $S'$ to size $k$ while achieving the set precision level. In such a situation, $p$ is increased by a step and the whole process repeats.

For the set of benchmarks studied, the pruning algorithm was always able to achieve $|S'| < k$. The average and the per-benchmark smallest and largest number of configurations retained after neighborhood based selection for CSSR are 11, 9, and 14 respectively.
The overall runtime for the algorithm is $O(kn^2)$, when $k$ configurations are needed to be retained. As $n$ is already restricted to a low value by $ELIC$, this does not pose a big problem. However, an additional optimization can be performed to speed up the algorithm, if deemed necessary. Since it is observed that configurations separated by larger differences in their indices are generally not connected, selecting them in parallel can lead to faster population of $T$. To achieve this, the list of configurations can be hashed into different buckets using modulo hashing scheme, and the pruning algorithm can then proceed to select configuration buckets into $S'$ in the above mentioned fashion rather than individually. However, we haven't implemented this strategy since $|S| - |S'|$ is small after $ELIC$ step.

4.5 Evaluation of the different CSSR pruning methods

4.5.1 Final configuration space

Figure 4.6 shows $|S'|$ following the three CSSR pruning methods. In the figure, x-axis shows the different benchmarks used for evaluation and y-axis shows $|S'|$. Merit based selection for CSSR always selects the target $k$ configurations based upon merit. Bound based selection for CSSR selects one configuration per interval bound. However, it is observed that $|S'|$ is slightly less than $k$. This happens because configurations with high values of $P_{norm}$ are sometimes selected for intermediate interval bounds which leads to unavailability of additional configurations to choose for the final few bounds. As expected, neighborhood based selection for CSSR aggressively prunes the configuration space and leads to small configuration spaces.

4.5.2 User demand tracking

A first order analysis is performed to quantify how the different methods for CSSR fare in satisfying user demands. Since user demands are variable, demands with different degree of performance and power requirements are considered. The different demand scenarios considered are shown in Table 4.5. For each scenario, a set of 10,000 user demands are synthetically generated. For each user demand, a single configuration from the retained configuration space that is expected to satisfy it is chosen. The inaccuracy in demand satisfaction is noted. The
Final configuration space size

Benchmark

merit based bound based neighborhood based

Figure 4.6 Final adaptive microarchitectural configuration space size

<table>
<thead>
<tr>
<th>Scenario</th>
<th>( P_d ) and ( W_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>High performance (HP)</td>
<td>( P_d &gt; 0.7, W_d = 1 )</td>
</tr>
<tr>
<td>Low power (LW)</td>
<td>( W_d &gt; 0.3, W_d &lt; 0.5, P_d = 0.3 )</td>
</tr>
<tr>
<td>Balanced demands (BD(\delta))</td>
<td>( P_d &lt; 0.7, P_d &gt; 0.5, P_d - W_d = \delta )</td>
</tr>
<tr>
<td>Stringent demands (SD(\delta))</td>
<td>( P_d &gt; 0.7, P_d - W_d = \delta )</td>
</tr>
</tbody>
</table>

Table 4.5 Different performance-power demand scenarios

inaccuracies in demand satisfaction for the different demands generated for a demand scenario are averaged to smooth out the effects of individual outliers.

Figure 4.7 shows the inaccuracies reported for the high performance demand scenarios. The x-axis in the figure denotes the different benchmarks and the y-axis reports the inaccuracy values as percentages (\( PI \)). Both bound based selection and neighborhood based selection track different user demands efficiently. The average \( PI \) reported for these selection methods is less than 1%. Merit based selection leads to an average of 9\% \( PI \). The increase in power consumption to boost performance at the higher end of the performance spectrum is typically higher than the corresponding increase at the low end of performance spectrum. As such, meritorious configurations are concentrated at the lower end of the performance spectrum. Many configurations at the higher end of performance spectrum are pruned by merit based selection which leads to unavailability of proper configurations to satisfy user demands in this
scenario.

Figure 4.8 shows the inaccuracies reported for low power scenarios. All the three CSSR pruning methods perform well in tracking user demands in this scenario. The slightly high $PI$ in the case of $astar$, $gobmk$, and $mcf$ can be attributed to the limitations of the considered adaptive controls. The least values of normalized power consumption possible for these benchmarks when using the configuration space retained after $ELIC$ step is 0.34, 0.35, and 0.33 respectively.

Figure 4.9 shows the inaccuracies reported for balanced demands when (a) $\delta = 0.1$, and (b) $\delta = 0.2$. As expected, the $PI$ values increase when $\delta$ is increased from 0.1 to 0.2. Once again, all the three pruning methods efficiently track the user demands. The increase in $PI$ between the cases of $\delta = 0.1$ and $\delta = 0.2$ is low for merit based selection compared to the other pruning methods. This shows how the merit metric used leads to selection of configurations providing higher performance while consuming lower power. The average $PI$ reported is less than 1% when $\delta = 0.1$ and is around 4% when $\delta = 0.2$. These lower values are an indication that the power needed to deliver intermediate performance levels is low.
Figure 4.9  $PI$ in tracking balanced demands

Figure 4.10 shows the inaccuracies reported for stringent demands when (a) $\delta = 0.1$, (b) $\delta = 0.2$, and (c) $\delta = 0.3$. It is observed that utilizing the configurations retained by bound based selection results in the lowest $PI$. The corresponding $PI$ values are closely followed by those obtained for neighborhood based selection. This shows the effectiveness of neighborhood based selection in simultaneously reducing $PI$s and configuration space size. Merit based selection results in large $PI$ values due to reasons explained earlier for the case of high performance demand scenario. Both bound based selection and neighborhood based selection lead to $PI$s which are less than 8% on an average when $\delta = 0.3$. Note that further reduction in $PI$ is possible if intra-application adaptation is considered.

### 4.6 Conclusion

In this chapter, a methodology to prune adaptive microarchitectural configuration space is presented. Using a x86 processor, it is first demonstrated that only a small set of adaptive hardware components is sufficient to achieve effective performance-power tradeoff. Next, it is shown that among the chosen adaptive components with different levels of adaptivity, only
Figure 4.10  $PI$ in tracking stringent demands
a small number of combinations (configurations) of them are meritorious to deliver the most
effective performance-power tradeoff. A set of algorithms are further designed to reduce the
number of configurations to a specified size to keep the run time complexity of utilizing them
low. Finally we make an observation that a small (16) pruned set configurations is effective
in satisfying varied user demands. The most effective pruning technique also depends on the
user needs of high performance, or low power or a balance between them.
CHAPTER 5. DEGRADATION OF PERFORMANCE-POWER TRADEOFF UNDER PERMANENT FAULTS

The adapted microarchitectural components and the glue logic providing the adaptivity are susceptible to permanent faults, like any other component on the microprocessor floorplan. A fault prohibits usage of a subset of adaptations originally provided. In Chapter 4, the configuration space has been significantly reduced to retain only the most useful configurations. Since all configurations retained are deemed important, it is necessary to evaluate how the tradeoff is affected when one or a few of the considered adaptations fail. This study also provides insight into how important each of the considered adaptive components and its associated adaptations are to effectively tradeoff performance and power. Our observations indicate that the change in system behavior in terms of performance delivered and power consumed due to occurrence of a fault typically stays below 10% of the required/desired levels when serving a large set of demands. When required performance cannot be provisioned, we observe that a significant power saving can be achieved as well. Additionally, we also narrow down to the adaptive controls that can be deliberately unused to conserve 5-7% additional power while sacrificing minimal performance when a fault occurs.

5.1 Introduction

The implementation of an adaptive hardware component requires extra logic circuitry to enable/disable the use of the various adaptive levels associated with it. This logic circuitry and the other building blocks in these adaptive components are both subject to permanent failure. The reason for the failure can be manufacturing defects, early life failures, or wear-outs ([22, 3, 11]). Wear-out failures can be further caused by one or a few of the following phenomena:
electromigration, stress migration, dielectric breakdown, etc. We analyze the system behavior when a permanent fault manifests in the hardware associated with the adaptive components. Since we select only the most effective hardware configurations for implementation, the presence of all these configurations may be required for effective performance-power tradeoff. This makes the failure analysis very important.

In the following analysis, it is assumed that the presence of the aforementioned faults is implicitly detectable. Analysis of fault detection methods is not within the scope of this research. It is assumed that faults are of permanent nature and are located and marked through existing fault detection mechanisms (ex. [9], [67], [10]). The reconfiguration is carried out so as to not utilize the faulty modules. The adaptation scheme then is only managing available components.

Integrity checking and fault detection cum tolerance techniques pertaining to cache memory already exist in commercial microprocessors [79]. Modern processors like the ARM cortex series of processors use a 64-bit ECC [62] to protect the instruction cache. To reduce the associated latency and power consumption, IBM Power 6/7 [83], AMD Opteron [59], SPARC64 [89], etc. protect the L1 instruction cache using parity. When the parity check detects an error, the instruction can be refetched. The non-stop architecture proposed in [8] provides a sniffer/scrubber that tests all memory locations for errors periodically. Errors are corrected if possible and written back. A reread followed by an integrity check can be used to decide whether the fault is transient or permanent. The memory locations containing permanent faults are taken out of service. Given the above, the only fault considered in a cache block is when the entire tag check logic fails for a single way. Thus, one way becomes inoperable. In such a case, cache will reduce from 8-way to 4-way associative in the considered fault scenarios. The considered faulty scenarios are detailed next.

5.2 Fault model

Analysis is performed to study the effectiveness of utilizing the pruned configuration space in catering to varied user demands when a single permanent fault occurs. The following fault scenarios are individually studied. Combinations of these faulty scenarios, though possible,
are not investigated in detail since such a fault manifestation probability becomes low. It will be noticed that when one component fails, to achieve proper performance-power balance, a possible adaptation corresponding to another adaptive component is not utilized any way. So effectively some scenarios of multiple faults in different control knobs are already implicitly covered. Further, absence of particular adaptive configurations may require the consideration of some configurations that were eliminated by the ELIC pruning step. Usage of such ineffective configurations may lead to lower $PI$ in case of faults. Hence, the entire configuration space retained after the SACK step is considered for this analysis.

**Fault scenarios**

1. Adaptive dispatch port failure: This scenario encompasses the failures associated with the functioning of a single dispatch port. The instructions scheduled for execution reside in FIFO buffers waiting to be issued for execution by the dispatch logic. Faults in the buffer can affect the functioning of the dispatch port. Faults in individual entries of the buffer can be handled using mechanisms discussed in [9]. The adaptive logic to enable/disable a dispatch port can encounter a failure at which point the dispatch port becomes inoperable. We study the system behavior under such failures. Since the allowed configurations for the dispatch width are 1, 2, and 4, failure of a dispatch port prohibits the use of a dispatch width of 4. Note that we assume that all the dispatch ports are homogeneous and all the ports can issue any arbitrary allowed instruction.

2. Adaptive cache way failure: Faults in individual memory locations can be dealt with parity, error correcting codes or other related techniques. Examples for these are provided later on. However, the tag comparator or the way enable logic for a single cache way can also fail. This prohibits the use of an associativity setting of 8. The allowed cache associativity levels when a cache way fails are 2 and 4.

3. Instruction window chunk failure: The instruction window can be designed as multiple chunks each of which can be enabled or disabled independently [13]. To provide for the considered adaptive configurations, the instruction window can be designed as 4 chunks,
each containing 32 entries each. When a single chunk fails, the deployable configurations are 32 and 64 entries. We do not consider the case of 96 entries, although possible under a single fault scenario, to keep all the adaptation sizes as power of 2. We consider faults that impede the operation of an entire instruction window chunk. Failures in individual entries can be masked using techniques proposed in previous research [9].

4. **VF setting failure**: Dynamic voltage and frequency scaling (DVFS) control is provided in real-world microprocessors as a 2-step process. Once the DVFS controller determines the correct VF setting to use, a control signal is sent to an on-board oscillator which adapts the clock frequency accordingly. Voltage scaling can be implemented in multiple ways [19]. The chip can be fed with multiple supply voltages and an individual voltage rail can be provided to carry each of these levels. A set of pull up transistors tap these voltage rails to provide supply voltage for the hardware components. The DVFS controller provides the necessary gate control signals for these transistors. A failure can occur either in the power rail (open circuit), the pull up transistor, or the input pin on the chip to which a voltage supply is connected. If the number of provisioned voltages is large, it becomes impractical to provide a large number of supply voltages (and power rails) to the chip. Alternatively, a DC-DC voltage converter can be provisioned on-board to generate the required voltage levels using a single input supply voltage. Requirement of large inductors and capacitors in this regard complicates the design. Since the chosen number of VF settings is small, we consider the former design practice. A single failure prohibits the adaptation of a single voltage and frequency level. The other two voltages and frequency levels can still be used.

Table 5.1 summarizes the different fault scenarios investigated and the available adaptation levels for the faulty components in the presence of faults.

**Potential fault detection schemes**  Current processors come with a set of counters and mechanisms to measure and store the on-chip voltage and frequency values. This infrastructure can be used to check the occurrence of desired DVFS transitions. Any discrepancy will indicate a failure of a certain voltage or frequency setting.
<table>
<thead>
<tr>
<th>No.</th>
<th>Fault scenario</th>
<th>Retained adaptations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dispatch port failure</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>2</td>
<td>Cache way failure</td>
<td>{2, 4}</td>
</tr>
<tr>
<td>3</td>
<td>Instruction window chunk failure</td>
<td>{32, 64}</td>
</tr>
<tr>
<td>4</td>
<td>Lowest VF setting failure</td>
<td>{(1.484 V, 1.6 GHz), (1.228 V, 1.2 GHz)}</td>
</tr>
<tr>
<td>5</td>
<td>Intermediate VF setting failure</td>
<td>{(1.484 V, 1.6 GHz), (1.036 V, 0.8 GHz)}</td>
</tr>
<tr>
<td>6</td>
<td>Highest VF setting failure</td>
<td>{(1.228 V, 1.2 GHz), (1.036 V, 0.8 GHz)}</td>
</tr>
</tbody>
</table>

Table 5.1 Investigated fault scenarios

Dispatch port failures are also easy to detect. The function of a dispatch port is to issue one or more selected instructions to available functional units. Further, the type of instruction dispatched will also determine which execution unit should process the instruction. As part of dispatch, proper inputs corresponding to the dispatched instruction are forwarded to the input wires of the functional unit. Comparing the actual inputs being fed to the functional unit against the expected inputs can determine if a dispatch occurred properly.

Additional cache tag comparators can be provided in the hardware and a triple modular redundancy based approach can be used for detection of cache tag comparator failure. Since the number of cache ways is usually small, we do not expect this to add significant hardware overhead.

For detecting failures in instruction window chunks, individual entries need to be monitored both before and after each update to ensure that the update happens as desired. Additional auxiliary registers can be provided to record the contents of individual entries before updates happen. The update can be mirrored in the additional registers using additional update logic. The contents of the auxiliary registers can be compared with corresponding instruction window entries after the update. Although this mechanism works for detecting faults, the associated hardware overhead will be significant. Additional studies need to be carried out to investigate alternate fault detection mechanisms.
5.3 Evaluation of tradeoff degradation

To quantify the system behavior under the presence of faults, the average performance delivered \( (P_{avg}) \), power consumed \( (W_{avg}) \), and the inaccuracy in satisfying the performance \( (Pin) \) and power constraints \( (Win) \) produced while catering to various user demands are analyzed. \( P_{avg} \) and \( W_{avg} \) are measured as fractions normalized to the maximal possible values (obtainable using \( S_{max} \)). The values for these metrics resulting from choosing an optimal configuration from the configuration space available with and without the presence of a fault are obtained and compared. The set of SPEC 2006 benchmarks considered for evaluating the different CSSR in Chapter 4 is used as workload for analyzing the system behavior in the presence of faults. The \( P_{avg}, W_{avg}, Pin, \) and \( Win \) reported are averaged over the corresponding values obtained in the case of these benchmarks.

Similar to the evaluation procedure in Chapter 4, user demands are synthetically generated to represent different demand scenarios. For each scenario, 10,000 user demands are generated. Each demand contains a primary as well as a secondary constraint. A single configuration that is deemed best to serve each user demand is chosen and the corresponding inaccuracy values are measured. For a demand, the chosen configuration satisfies the following properties.

1. The chosen configuration satisfies the primary constraint. If no available configuration satisfies the primary constraint, the configuration leading to the lowest \( PI \) in satisfying it is selected.

2. If primary constraint is satisfied by multiple configurations, the chosen configuration performs the best with regards to the secondary constraint.

For all expect low power demands, performance is considered the primary constraint. The tradeoff degradation for these 10,000 demands is averaged to eliminate individual outliers.

5.3.1 Dispatch port failure

Figure 5.1 ((a) and (b)) shows how the system behavior changes/degrades when one of the dispatch ports or the associated adaptive logic fails. In the figure, the x-axis denotes the user
Figure 5.1 Tradeoff degradation when one dispatch port fails demand and fault scenarios as $mode_{scene}$ where $mode$ represents the user demand scenario and $scene$ tells whether a fault exists $f$ or not $all$. The y-axis denotes the (a) demand satisfaction inaccuracies ($Pin$ and $Win$), and (b) the delivered average performance ($P_{avg}$) and consumed power ($W_{avg}$).

Observations.

1. An 4% decrease in performance is observed when serving high performance demands. Although the microprocessor is limited to dispatching a maximum of two instructions per cycle, utilization of high voltage and frequency guarantees that the performance stays reasonably high. However, provision of the demanded high performance requires the utilization of more dispatch ports. It is also noticed that the power consumption stays fairly similar with or without the fault. This happens since power inefficient configurations are selected to meet the performance demand when a fault occurs. Hence, more power consumption is observed while providing lesser performance than demanded.

2. The performance loss observed when serving the low power demands is just $\sim$2%. Higher dispatch widths are usually used only for catering to stricter performance requirements, and the absence of the associated configurations does not affect low power user demands. For the few cases when a higher dispatch width paired with minimal configurations for the other adaptive units can be used to serve low power demands, absence of such con-
figurations necessitated the use of other ineffective configurations to satisfy the primary constraint. This resulted in the observed performance loss.

3. When serving balanced user demands, the power consumption is traded off slightly to satisfy the performance constraint. It is observed that the increase in power consumption (compared to the all scenario) is $\sim 3\%$.

4. The power constraints in the stringent demands are satisfied inaccurately. However, such an inaccuracy is also observed for the all scenario as well. The difference between the $Win$ values for the all and $f$ scenarios is negligible. The average increase in $Pin$ due to failure is capped at 3.5%.

5. Overall, loss of the wider dispatch width adaptation can be reasonably compensated for by choosing alternate configurations except when performance demands are at the peak level. We found no one-to-one correlation between the faulty configurations (otherwise employed in all scenario) and the alternate configurations chosen in place of these.

5.3.2 Cache way failure

Figure 5.2 shows how the tradeoff changes/degrades when one of the cache ways fails. The axes in this figure as well as the figures following this in the next few subsections investigating individual failure scenarios follow the notations as in Figure 5.1. Note that failure of a cache way still theoretically retains 7 other possible settings (1-7). Since we have restricted our adaptations to just 2, 4, and 8 ways (powers of 2), we proceed to choose only configurations that utilize 2 or 4 cache ways in case of a cache way failure. If the resultant tradeoff is found to be significantly suboptimal to the case when no fault occurs, we can later opt to consider additional cache adaptations in the future.

Observations.

1. We observe that the tradeoff under the presence of a fault is strikingly similar to the behavior when all adaptive controls are usable. In particular, the primary constraints are always satisfied as per the requirement.
2. The observed differences between the delivered performance and consumed power between the fully adaptive and the faulty scenarios are always less than 1%. However, the limitations with the accuracy of the simulation framework used prohibit us from accurately commenting on the fine differences between the observed performance and power between the two scenarios.

3. Based on these observations, we claim that investigation of additional levels of adaptivity for the instruction cache may not be required. The other adaptive controls included into the configuration space are enough to compensate for the loss of the maximum adaptive level for the instruction cache.

### 5.3.3 Instruction window chunk failure

Figure 5.3 shows how the tradeoff changes/degrades when one of the instruction window chunks fails. Since adaptation of instruction window has the lowest tradeoff potential, we expect the system to degrade only marginally when a fault occurs.

**Observations.**

1. A slight inaccuracy (∼1.5%) manifests in serving the performance constraint in the high performance demands. A similar case also holds for stringent user demands. The maximum possible instruction window size becomes the performance bottleneck. To deliver
Figure 5.3  Tradeoff degradation when an instruction window chunk fails

the required performance, a number of aggressive cum ineffective configurations which are otherwise eliminated by the ELIC pruning step are now utilized. This results in a slight increase in the power consumption when compared to the all scenario for the stringent user demands.

2. Low power demands are still satisfied well. Lower instruction window sizes, which are typically used to serve the associated demands, are still available for selection.

3. When serving balanced user demands, power is traded off slightly to satisfy the performance as per the requirement. It is observed that the increase in power consumption (compared to the all scenario) is $\sim 2\%$ when $\delta = 0.2$.

4. In general, loss of the highest IW setting leads to a slight loss of performance as well as a slight increase in power consumption when serving high performance requirements.

5. From these observations, we can claim that consideration of the currently unused but theoretically possible instruction window adaptation size (96 entries) cannot guarantee significantly better system behavior.

5.3.4  Voltage and frequency control failure

Since the different VF controls can independently fail, we analyze the system behavior under the possible faulty scenarios separately.
5.3.4.1 Failure of the lowest $VF$ setting

Figure 5.4 shows how the tradeoff changes/degrades when the lowest $VF$ setting becomes unusable. We make the following observations.

**Observations.**

1. Since the lowest $VF$ setting is almost never used to serve high performance user demands, the system behavior while serving these demands remains unchanged under this failure.

2. The observed inaccuracy in serving the power constraint in the low power demands is just $3\%$. Choosing an intermediate $VF$ setting while lowering the dispatch width, instruction cache associativity, and instruction window size resulted in maintaining a similar power profile (as that produced in all scenario). A $2\%$ loss of performance is also observed in this case, since other ineffective configurations are now utilized to satisfy low power demands.

3. The tradeoff behavior while satisfying balanced and stringent user demands remains reasonably unchanged ($\sim 1\%$ deviation).
5.3.4.2 Failure of the intermediate $VF$ setting

The changes in tradeoff behavior noted when the intermediate setting for $VF$ control fails is shown in Figure 5.5. We make the following observations under this fault scenario.

Observations.

1. The highest $VF$ setting is almost always required to satisfy high performance demands and the intermediate $VF$ control is sparsely used under such circumstances. Since the highest $VF$ control is still active, the $PI$ when serving such demands remains unchanged. Due to the discrete nature of the provided performance and power values, an increase is both $P_{avg}$ (1%) and $W_{avg}$ (3%) is observed when the fault manifests.

2. The effects of trading off the secondary constraint to satisfy the primary constraint are quiet noticeable when the intermediate $VF$ setting fails. The primary constraint is always satisfied while the degradation in system behavior with respect to the secondary constraint varies between 1% and 12% when serving balanced and stringent user demands. This effect is more pronounced for the case of balanced user demands.

3. Since the obtainable performance and power characteristics are discrete in nature, the exact satisfaction of the primary constraint is not always possible. This leads to the selection of configurations that provide for the primary constraint well above the requirement.
5.3.4.3 Failure of the highest $VF$ setting

The change/degradation in tradeoff noted when the highest setting for $VF$ control fails is shown in Figure 5.6. We make the following observations under this fault scenario.

**Observations.**

1. A 7.5% shortfall in performance manifests in the satisfaction of high performance and stringent user demands. The utilization of an intermediate $VF$ setting coupled with an aggressive configuration of the other adapted components is insufficient to provide for the high performance requirements. However, the system behavior degrades gracefully and the inaccuracy in demand satisfaction is not very large. An average of 10% decrease in power consumption is observed as well, when performance constraint cannot be satisfied.

2. The tradeoff remains strikingly similar to the scenario where all adaptive controls are active when serving low power and balanced user demands.
5.3.5 Power saving with reduced performance requirements

While investigating single component failures, we have encountered two different effects to the performance-power balance when demands containing intermediate to higher performance constraints are posed. When the performance requirement can be satisfied using alternate ineffective configurations (when the ideal configuration fails), an associated increase in power consumption is noticed. When such a possibility does not exist, a lower performance is delivered. In this case, a reduction in power consumption is noticed as well. We next investigate the sensitivity of each adaptive component’s failure towards provisioning different degrees of performance and note the power consumption. The results presented in this regard also provide insights to the user about the achievable power reduction when performance demands are mellowed down. The observed performance and power consumption values reported are averaged over the set of benchmarks that are being investigated so far.

Table 5.2 shows the $P_{avg}$ and $W_{avg}$ values when the performance demanded is set to 90%, 80%, and 70% separately. In the table, none refers to the scenario where all adaptive components are active.

<table>
<thead>
<tr>
<th>Failure</th>
<th>Performance requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90%</td>
</tr>
<tr>
<td>None</td>
<td>94.2</td>
</tr>
<tr>
<td>Dispatch port</td>
<td>86.7</td>
</tr>
<tr>
<td>Cache way</td>
<td>93.6</td>
</tr>
<tr>
<td>Instruction window</td>
<td>90.8</td>
</tr>
<tr>
<td>Low VF</td>
<td>94.7</td>
</tr>
<tr>
<td>Intermediate VF</td>
<td>94.3</td>
</tr>
<tr>
<td>High VF</td>
<td>79.3</td>
</tr>
</tbody>
</table>

Table 5.2 Performance and power characteristics obtained for different performance demands

Observations.

1. We observe that the performance requirement of 90% is deliverable unless a dispatch port or the highest VF setting fails.
2. The maximum deliverable performance when a dispatch port fails is 86.7%. Similarly, the maximum deliverable performance when the highest VF setting fails is 79.3%. When a 90% performance demand is encountered in these situations, the observed power saving is 23.8% and 36.6% respectively.

3. 80% and 70% performance demands are always closely satisfied under faulty scenarios.

4. Since the configuration space is discrete, there arise situations when performance is over-provisioned just to stay above the required level. It is noticed that this over-provisioning never exceeds 5.8% for the set performance constraints.

5. When 90% performance is demanded, the power saving obtained under the fault conditions where the performance demand can be satisfied is at least 19.5%.

6. When 80% performance is demanded, the power saving obtained is around 30%. The power saving obtained while satisfying the performance constraint is nearly equal in the cases of all single component failures, except when the highest VF setting fails. When the highest VF setting fails, the reduction in power seen is higher (∼39%). The other fault scenarios overcompensate for performance and lose some achievable power gain.

7. When 70% performance is demanded, the power saving obtained is 35.5-47.5%.

5.3.6 Avoiding available adaptations for increased power saving

The microprocessor design process is inclusive of the analysis of synergy between the maximal configurations provided for the different hardware components. The failure of the most aggressive configuration for an adaptive component can result in disruption of this synergy for the available configuration space. Such situations can be analyzed and exploited to save more power. Consider two adaptive components $i$ and $j$. Let the maximum performance obtainable (and power consumed) when the most aggressive configuration for $i$ fails is $P'_i$ ($W'_i$). Similarly, let the maximum performance obtainable (and power consumed) when the most aggressive configuration for $j$ is additionally disabled/not considered for adaptation is $P''$ ($W''$). If $P''$ is close to $P'_i$ but $W'' \ll W'_i$, we can choose to not consider the most aggressive configuration for
Figure 5.7 Deliverable peak performance and the associated power consumption utilizing a subset of available configuration space.

1. When the maximal setting for an adaptive control other than instruction cache fails, it is observed that further ignoring the instruction cache set associativity setting of 8 results in significant power saving, while limiting the performance sacrifice. The obtained power savings when the highest setting for \( VF, IW, \) or \( DW \) fails are 5\%, 7\%, and 7\% respectively, while the performance losses are limited to 1\%, 2.5\%, and 1\% respectively. The increase in execution time due to bottlenecks in the failed controls overshadows the additional execution time due to the extra cache misses caused.
2. When a failure occurs in an instruction window chunk, instruction cache way, or dispatch port, further disabling the highest $VF$ setting provides a 25-30% power gain. However, a significant performance loss (about 20%) is also observed in these cases.

3. Other possibilities of disabling the aggressive setting for an adaptive component when a failure occurs in any other component are not beneficial as well.

5.4 Conclusion

In this chapter, we analyzed the effectiveness of a pruned adaptive microarchitectural configuration space in effectively trading off performance and power when a permanent fault occurs. The presence of a fault in an adaptive component necessarily prohibits the use of a subset of the originally provided configurations. For the considered adaptive controls except ($VF$), the permanent fault prohibits the use of the associated maximal configuration. For the $VF$ control, the fault prohibits the use of a single $VF$ combination. We observed that the change in the system behavior in terms of performance delivered and power consumed due to occurrence of a fault typically stays below 10% when serving a large set of varying demands. At least 90% of the original performance can still be delivered when a fault occurs in an instruction window chunk, instruction cache way, or the lower or intermediate $VF$ setting employed. In other faulty situations, the obtainable maximum performance stays above 80%.

When an adaptive control setting that leads to system operation in a particular region in the performance-power spectrum fails, alternate configurations are chosen to satisfy demands associated with this region of the spectrum. The satisfaction of secondary constraint in the user demand is traded off more aggressively to satisfy the primary constraint when compared to the scenario where no fault occurs. In particular, this effect is more pronounced ($\sim 10\%$) when a fault renders the intermediate $VF$ setting useless. We have observed that an inaccuracy in satisfying the primary constraint rarely manifests. Under some fault scenarios, extreme high performance demands cannot be satisfied accurately. In such cases, a drop in power consumption is observed as well. We note that at least 80% of the maximal performance can still be delivered when a fault occurs, while saving at least 30% power.
Finally, we noticed that the use of high associativity (8) in the instruction cache doesn’t produce any noticeable performance benefit when a fault occurs that affects the most aggressive setting in any other adaptive control. The performance advantage provided by the high associativity is overshadowed by the performance loss caused by the failed adaptive control. Hence, the adaptations inclusive of this associativity setting for cache need not be considered once a fault is detected. This leads to an additional 5-7% power saving.

From our observations, we conclude that the considered adaptive configuration space produces system behavior which is generally resilient to a single permanent fault. Failure of configurations corresponding to a single control setting leaves out alternate configurations (which are still operable) that guarantee similar system behavior.
CHAPTER 6. APPLICATION AWARE PERFORMANCE-POWER TRADEOFF

This chapter covers the details of the different adaptation strategies implemented for microprocessor performance-power tradeoff. An alternate classification of previously proposed adaptation strategies is introduced. The advantages and disadvantages associated with various adaptation strategies is presented. The details of our comprehensive static cum dynamic, as well as dynamic only adaptation strategies follow this discussion. A detailed evaluation is carried out to analyze the effectiveness of the developed adaptation strategies in utilizing the pruned configuration space to provide the required performance-power tradeoff in different regions of the possible performance-power spectrum. In most cases, it is observed that these demands on performance and power are satisfied with up to 90% accuracy. For a set power level, the obtained performance leveraging on our pruned configuration space is found to be 7% lower compared to a state-of-the-art scheme using 10 times the configuration space. It is also observed that the use of the developed dynamic adaptation strategies leads to energy efficient execution. The observed energy efficiency is close to 95% of the ideal efficiency obtained with a comprehensive oracular adaptation scheme.

6.1 Introduction

It is a well-known fact that hardware-software interactions vary during application execution. For example, the exhibited parallelism depends on the instruction stream, and as a consequence, wide variation in ALU or cache usage exist. The application execution profile is generally demarcated by phases/intervals with different execution characteristics. It is important to exploit these changes in application behavior and find the best configuration for each
phase. This serves two purposes. First, it results in better tradeoff decisions since each chosen configuration will be tailor fit to the associated program phase. Second, due to the discrete nature of performance-power points provided by the configuration space, a single configuration may not be sufficient to exactly satisfy different demands. However, a composition of a few configurations works well.

Microarchitectural adaptation strategies can further be classified into static \([44, 65, 29]\) and dynamic \([61, 31, 2]\) strategies based upon when the adaptation decisions are made. Static strategies assume the knowledge of the entire application execution characteristics and statically find a configuration to use per each application phase. Since decisions are static, they can be comprehensive in nature since the overhead does not fall within execution time. Also, the knowledge of future application phases makes the adaptation decisions optimal. Such strategies have two limitations. First, the assumed knowledge of the execution profile may not be practical for general purpose computing platforms. Second, the assumed knowledge may become invalid due to runtime variations. Such strategies are generally suited for real-time systems and HPC platforms where execution time is quite predictable. A dynamic strategy monitors specific execution characteristics at runtime and deploys suitable configurations accordingly. Since the adaptation analysis is performed at runtime, it is generally restrictive in nature to avoid impractical overhead. Also, the analysis is limited to peephole of instructions and the adaptation decisions are suboptimal. In Section 6.2, the details of our static cum dynamic strategy for performance-power tradeoff are presented. The application execution profile is divided into phases and configurations are chosen for deployment during the individual phases statically. A simplistic runtime manager adjusts these static decisions to meet runtime requirements on performance and power consumption. The static component ensures tradeoff optimality and the dynamic component adapts the adaptation decisions to observed runtime variations. Two lightweight dynamic only adaptation strategies are also developed that leverage on the small size of the configuration space and make quick adaptation decisions. The details of these strategies are presented in Section 6.3.
6.2 Two stage static cum dynamic adaptation strategy

6.2.1 Application phase demarcation

Consider an application whose execution profile contains $I$ instructions. A phase generator ($PG$) divides this into $M$ intervals, a process that will be referred to as Phase generation ($PGen$), where phase $i$ contains $I_i$ instructions. The idea is to initiate a microarchitectural adaptation once after each phase finishes execution. The process can be represented as $< I_1, I_2, ..., I_M >= PGen(I, HSIP) \mid \{I_i \neq 0, \sum_i I_i = I\}$. In this representation, $HSIP$ denotes the hardware-software interaction patterns. A $HSIP$ is a 2-tuple $<IPS, W_a>$, where $IPS$ denotes the average instructions committed per nanosecond, and $W_a$ denotes the average power consumption, in Watts. $HSIP$s are collected at the granularity of 1 million instructions each to avoid impractical profiling overhead.

The value of ($M$) directly affects the tradeoff optimality. A large $M$ implies a large overall runtime overhead, while providing fine-grained adaptations. On the other hand, a small $M$ restricts adaptation control, as the number of adaptation opportunities becomes limited. Our strategy is to necessitate reconfiguration when either 1) the $HSIP$ changes significantly, or 2) any application phase gets very long. Since $HSIP$ are collected at the granularity of 1 million instructions, it also constitutes the interval length, the minimum length of instructions separating two adaptation instances. Incidentally, it has been noticed later that this guarantees that the runtime adaptation overhead is always $< 1\%$, which is explained later.

The phase generator divides the application execution profile into intervals of 1 million instructions each, and marks phase changes at the end of selected intervals. The achievable $IPS$ spectrum utilizing the allowed architectural configurations is first divided into 20 equal sized intervals. This implies that adjacent intervals are centered on $IPS$ values which are separated by 5% of the maximum $IPS$. A similar division is made for the $W_a$ spectrum as well.

A set of 400 buckets are generated from these windows, each of which covers a specific range of $IPS$ and $W_a$ values. The application execution with $S_{max}$ leads to $IPS$ and $W_a$ contained by a single bucket ($FindBucket$ function in Algorithm 3). A phase change is demarcated when
the associated buckets for two adjacent intervals are different. A phase change demarcation also occurs when the current phase length becomes larger than 10 million instructions (Re- condChange function in Algorithm 3). The latter demarcation strategy is especially useful in cases where HSIP remain invariant. In such cases, employment of a single configuration for the entire execution may lead to poor demand satisfaction due to the discrete nature of provided performance-power spectrum. The working of the phase generator for an application with $n$ million instructions is shown in Algorithm 3.

<table>
<thead>
<tr>
<th>Algorithm 3 Phase generation algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{sep} \leftarrow 0.2$</td>
</tr>
<tr>
<td>$w_{sep} \leftarrow 1$</td>
</tr>
<tr>
<td>$bkt_prev \leftarrow 0$</td>
</tr>
<tr>
<td>$p_len \leftarrow 0$</td>
</tr>
<tr>
<td>$i \leftarrow 1$</td>
</tr>
<tr>
<td>while $i \neq n$ do</td>
</tr>
<tr>
<td>$bkt \leftarrow \text{FindBucket}(interval_i)$</td>
</tr>
<tr>
<td>if $bkt \neq bkt_prev$ or $p_len = 20$ then</td>
</tr>
<tr>
<td>$\text{RecordChange}(i)$</td>
</tr>
<tr>
<td>$bkt_prev \leftarrow bkt$</td>
</tr>
<tr>
<td>$p_len \leftarrow 0$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$p_len \leftarrow p_len + 1$</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>

The first stage of our adaptation strategy is called the static reconfiguration stage. In this stage, the phase-wise performance-power consumption characteristics, jointly referred to as PWC, are collected and are recorded into a Stats database. The performance (power) is stored in terms of execution time (wattage). A set of configurations deemed best for the different application phases (one per phase) are selected by the Static configuration generator (SCG) to satisfy the overall demand. The information generated in this stage is passed through a Metadata database onto the second stage, namely the dynamic reconfiguration stage.

The second stage of adaptation utilizes a novel lightweight Runtime manager (rm) that selectively alters the deployed configurations from those chosen by SCG for each application phase locally. These alterations are meant to account for any deviation between the expected
and actual $PWC$ values aggregated over the past phases. Figure 6.1 shows the architectural adaptation process. The next two sections describe the two stage adaptation strategy in detail.

### 6.2.2 Static reconfiguration stage

Consider an application execution profile that contains $m$ phases. Let there be $n$ allowed configurations. Also, let the expected time (power) consumed for executing phase $i$ with $C_j$ be given by $t_{ij} \ (w_{ij})$. The set of all $t_{ij}$s ($w_{ij}$) can be represented using a single $nm \times 1 \ (1 \times nm)$ matrix $T \ (W)$, such that the $k^{th}$ row (column) of $T \ (W)$ contains $t_{(k/m)(k%m)} \ (w_{(k/m)(k%m)})$. $T$ and $W$ are computable from the phase info and offline profiling discussed in Section 6.2.1.

The problem of demand satisfaction is formulated as an optimization problem involving binary variables $s_{ij}$. These variables represent the usage of the allowed configurations for different phases. The utilization (and non-utilization) of $C_j$ for the phase $i$ is represented by $s_{ij} = 1 \ (s_{ij} = 0)$. All $s_{ij}$s are collectively represented using a $nm \times 1$ selection matrix $S$. The element in $k^{th}$ row of $S$ is given by $s_{(k/m)(k%m)}$. The following conditions are implicit.

\[
\begin{align*}
\{s_{ij} = 0 \lor 1\} & \forall \{i \in [1, m], j \in [1, n]\} \\
\sum_{j=1}^{n} s_{ij} = 1 & \forall \{i \in [1, m]\}
\end{align*}
\] (6.1)
The last condition can also be formulated as \( M \times S = K \), where \( M \) is a mask matrix of dimensions \( mn \times mn \). The matrix \( M \) is formulated such that in the \( i^{th} \) row of \( M \), elements \((i - 1) \times n + 1\) to \( i \times n \) are set to 1, and the rest are set to 0. \( K \) is a \( nm \times 1 \) unit matrix. Utilizing this representation combines the constraints for selecting one configuration per phase over all the phases into a single equation.

Using these notations, the net time \((T_{\text{net}})\) and average power consumed \((W_{\text{avg}})\) for the execution can be calculated as

\[
T_{\text{net}} = S' \times T, \quad W_{\text{avg}} = \frac{S' \times (T \times W')}{S' \times T} \tag{6.2}
\]

where \( A' \) represents the transpose of a matrix \( A \).

Different priorities (primary and secondary) are allotted to the satisfaction of \( P_d \) and \( W_d \) as part of \( U_d \). This distinction enables solutions that satisfy at least the primary constraint when satisfaction of both is impossible. The \( SCG \) targets the exact satisfaction of the primary constraint, while optimizing system behavior with respect to the other. The secondary constraint is implicitly optimized, even if there is no explicit demand on it. Assignment of priorities to constraints leads to two classes of problems, which are detailed next.

### 6.2.2.1 \( U_d \) with primary constraint on \( W_d \)

In this situation, the minimization of \( T_{\text{net}} \) is targeted subject to a hard bound \((W_{\text{bound}})\) on \( W_{\text{avg}} \). \( W_{\text{avg}} \) is considered the primary constraint rather than energy consumption \((E_{\text{net}})\). The latter can favor solutions that adopt the \( \text{max} \) configuration, which leads to higher \( W_{\text{avg}} \) and reduced reliability. Note that hardware reliability is inverse exponentially proportional to the chip temperature, which follows the trend of \( W_{\text{avg}} \). It could be argued that such solutions provide an opportunity to move the processor into a deep sleep state after quick execution. However, it is assumed that tasks are generally present at all times in the task queue, negating the possibility of drifting to deep sleep states. This makes our solution very amenable to data centers running batch jobs. The problem can be formulated as the following Integer linear program \((ILP)\) and solved.
minimize $S' \times T$

subject to

$$\frac{S' \times (T \times W')}{S' \times T} < W_{\text{bound}}, \ M \times S = K$$

The constraint on $W_{\text{avg}}$ can be transformed as follows.

$$S' \times (T \times W'_s) < 0 \text{ where } W_s = W - W_{\text{mat}}$$

(6.4)

Here, $W_{\text{mat}}$ is a $1 \times mn$ matrix with each element being $W_{\text{bound}}$ and the operation ‘$-$’ represents element-wise subtraction for two matrices. This transformation makes the problem a convex binary ILP. Several methods for solving such problems exist, e.g., cutting planes [58], branch and bound [86], branch and cut [95], and heuristics like tabu search [36], hill climbing [54], simulated annealing [98], etc. The extensively used branch and cut method [103] is utilized due to its advantage of combining the optimality provided by cutting planes and time efficiency provided by branch and bound solutions. Further details on this method are not provided, since it is not a part of our research contribution. The solution $S$ found by solving the ILP in turn yields the best configurations selected for the different application phases.

### 6.2.2.2 $U_d$ with primary constraint on $P_d$

In this situation, the minimization of $W_{\text{avg}}$ is targeted subject to a hard bound ($T_{\text{bound}}$) on $T_{net}$. The problem can be formulated as follows.

$$\text{minimize } \frac{S' \times (T \times W')}{S' \times T}$$

subject to

$$S' \times T \leq T_{\text{bound}}, \ M \times S = K$$

(6.5)

It can be immediately observed that the objective function is not convex and linear. Several approaches have been proposed in literature ([14], [41], [21], etc.) to tackle such problems. The widely utilized interior point algorithm is employed to select appropriate configurations for different application phases.
6.2.2.3 Storage overhead of Metadata database

As mentioned earlier, information generated during static reconfiguration stage is passed onto dynamic reconfiguration stage through the Metadata database. In this section, the storage overhead associated with this database is quantified. Four pieces of information are stored in the Metadata database.

1. The phase changes list (PCL). Precisely, the serial numbers of the intervals after which phase changes are demarcated are passed on. If there are $m$ phases, this data occupies $m \times \lceil \ln m \rceil$ bits, since $\lceil \ln m \rceil$ bits are required to encode a single phase id. For convenience of storage and retrieval, it is assumed that each id is actually stored as a short integer (16 bits), limiting the value of $m$ to $2^{16}$. In such a case, the storage required for PCL is $2 \times m$ bytes. Limits on $m$ can be imposed by reiterating the phase demarcation algorithm while increasing $p_{sep}$ and $w_{sep}$ in steps, until the number of resultant phases are less than $m$.

2. The statically selected best configuration list (BCL). If there are $n$ allowed configurations, $\lceil \ln n \rceil$ bits are required to uniquely represent a single configuration. Hence, a total of $m \times \lceil \ln n \rceil$ bits are required. Similar to the previous case, it is assumed that each configuration id is represented using a single byte. In such a case, the storage required for BCL is $m$ bytes.

3. The expected aggregated PWC (simply referred to as expected PWC from here on) after each application phase. Timing values are aggregated over multiple application phases by simple addition. The average power is aggregated by weighted arithmetic mean, where times for the different application phases act as weights. Since performance and power can be represented using floating point values (64 bits each), the net storage for expected PWC values is $16 \times m$ bytes.

4. The relative ordering of the allowed configurations, separately for performance (Performance order list- POL) and power (Power order list- WOL). This needs a total storage of $2 \times m \times n$ bytes.
The total storage overhead ($S_{over}$) can be calculated as

$$S_{over} = 2 \times m + m + 16 \times m + 2 \times m \times n \text{ bytes} = 19 + 2 \times n \text{ bytes} \quad (6.6)$$

Since the maximum value of $n$ is 16, $S_{over}$ is bounded to $51 \times m$ bytes. When $m$ is further restricted to $2^{16}$ (example), $S_{over} = 3.18$MB. Note that the entire database need not be cached at all times, since there is no temporal locality. Intelligent prefetching allows the data to flow into the cache seamlessly. The real cache footprint for this data corresponds to the data stored per phase, which is just 51 bytes.

6.2.3 Dynamic reconfiguration stage

6.2.3.1 Working on runtime manager

A lightweight runtime manager ($rm$) is developed to handle runtime $PWC$ variations after each application phase. The working of the runtime manager is illustrated in Algorithm 4. The runtime manager observes the slack in the schedule with respect to the primary constraint and selectively alters the statically chosen configuration for the next phase. Consider its invocation after execution of phase $i$. In the following, the statically selected configuration for phase $i + 1$ is referred to as the preselected configuration, and the alternate configuration chosen by the runtime manager as the reselected configuration.

A Performance monitor ($PowerMonitor$) is employed to measure the absolute time (average power) consumed by the application so far ($ReadPerformanceMonitor$ and $ReadPowerMonitor$ functions in Algorithm 4). Such functionality can be easily availed using hardware existent in modern processors, e.g. Intel Sandybridge family. These measured values together represent the actual $PWC$. The expected $PWC$ values are retrieved from the $Metadata$ database using $ReadMetaPerf$ and $ReadMetaPow$ functions. A significant difference ($\sim 5\%$) between the expected and actual $PWC$ values for the primary constraint, calculated by $CalculateSlackDirn$ in Algorithm 4, triggers configuration reselection process for phase $i + 1$.

The preselected configuration for phase $i + 1$ is read from $BCL$ ($ReadMetaConfig$ in Algorithm 4). The relative ordering of the allowed configurations in terms of the primary constraint satisfaction for phase $i + 1$ is read from $OL$ (either $POL$ or $WOL$, according to primary con-
Algorithm 4 Runtime reconfiguration algorithm

\[
\begin{align*}
\text{actual perf} &= \text{ReadPerformanceMonitor}(); \\
\text{actual pow} &= \text{ReadPowerMonitor}(); \\
\text{expected perf} &= \text{ReadMetaPerf}(i); \\
\text{expected pow} &= \text{ReadMetaPow}(i); \\
\text{static config} &= \text{ReadMetaConfig}(i + 1); \\
\text{slack dirn} &= \text{CalculateSlackDirn}(\text{expected}, \text{actual}); \\
\text{if} \; \text{slack dirn} = \text{prev slack dirn} \; \text{then} \\
& \quad \quad \text{Step size} \leftarrow \text{Step size} + 1 \\
\text{else} \\
& \quad \quad \text{Step size} \leftarrow 1 \\
\text{end if} \\
\text{prev slack dirn} &= \text{slack dirn} \\
\text{pos} &= \text{ReadMetaPos}(\text{OL}, i + 1, \text{preselected config}); \\
\text{if} \; \text{slack dirn} = \text{positive} \; \text{then} \\
& \quad \quad \text{pos} = \text{pos} + \text{Step size} \\
\text{else} \\
& \quad \quad \text{pos} = \text{pos} - \text{Step size} \\
\text{end if} \\
\text{AdjustPos}(); \\
\text{reselected config} &= \text{FindMetaConfig}(\text{OL}, i + 1, \text{pos}); \\
\end{align*}
\]

The \( rm \) also tracks the slack development trend. This information is used to tune the aggression in reselecting alternate configurations for the future application phases. A special variable \textit{step size} is used to dictate this aggression. A \textit{step size} value of \( k \) implies that the preselected and reselected configurations are separated by \( k \) entries in the \( OL \). The position of the preselected configuration is read off from the \( OL \) using the \text{ReadMetaPos} function. The new position for the reselected configuration is calculated using the slack direction and \textit{step size}, and the reselected configuration is read from \( OL \) using \text{FindMetaConfig} function. The \text{AdjustPos} function ensures that the position of reselected configuration in the \( OL \) is valid.

At the beginning of application execution, \textit{step size} is set to 1. Monotonic appearance of slack in a single direction increases \textit{step size} by 1. If the slack changes direction, \textit{step size} is reset to 1. The overall runtime for the algorithm is \( O(n) \). Since \( n \) is limited to a small value (16), the \( rm \) operates in constant time.
6.2.3.2 Runtime management overhead

As the \( rm \) is invoked after each phase during execution, its runtime overhead needs to be factored into the overall execution time. The worst case runtime for \( rm \) during execution is given by

\[
T_{\text{overhead}} = (m - 1) \times (t_{rm} + t_{reconfig})
\]

(6.7)

where \( t_{rm} \) denotes the time required to reselect a configuration, \( t_{reconfig} \) denotes the time required to perform the desired reconfiguration in hardware, and \( m \) denotes the number of application phases. Previous research has indicated that \( t_{reconfig} \) is low. If it is ensured that \( t_{rm} \) is low compared to all \( t_{ij} \)s, a low value for \( T_{\text{overhead}} \) is automatically guaranteed. This eliminates the need to consider \( T_{\text{overhead}} \) during static configuration selection, since small deviations in \( PWC \) can be handled efficiently during runtime. It is observed that the ratio \( \frac{t_{rm}}{t_{ij}} \) is less than 1% for all \( i \)s and \( j \)s, considering a minimum phase length of 1 million instructions.

Following this observation, the minimum phase length is fixed at 1 million instructions.

6.3 Dynamic adaptation strategies

Two factors motivate dynamic adaptation methodology. First, it may be too cumbersome to obtain phase-wise \( PWC \) values. As such, the static reconfiguration stage may not be practical for all computing scenarios. Also, such characteristics may vary with program inputs and values obtained through offline profiling may not hold well. Second, the size of the Metadata database becomes a factor of concern as larger \( m \) values are allowed. To avoid these concerns, two simplistic dynamic adaptation strategies employing our already developed runtime manager are proposed in this research. Since it is decided to avoid storing the expected \( PWC \) values for each program phase, an alternate mechanism to investigate the performance and power consumption is required. It is chosen to expect constant \( TPI \) (time per instruction) and power profiles for the entire execution. Note that this is not ideal due to the fact that performance and power consumption vary frequently during execution. As such, this constraint has a pitfall of not being able to exploit information from future application phases and optimize the tradeoff accordingly. However, this constraint lets us develop simple dynamic adaptation strategies.
Two variations of the dynamic only adaptation strategies are considered. For both these dynamic strategies, a single POL (or WOL) is used for the entire application. This list is obtained through offline profiling. The CPI and watts consumed with $S_{max}$ is similarly obtained. These values are scaled according to the normalized components in the demands to produce expected TPI and power consumption values. Based on the normalized demands, a single configuration that is expected to satisfy them best is chosen. This can be obtained through offline profiling as follows. The application is executed with the different allowed configurations one after the other and the expected time and average wattage are recorded. These values are normalized with respect to the corresponding values obtained for the execution with $S_{max}$. Any input demand can then be checked against this list of normalized values to choose the configuration that best satisfies it.

The application execution is demarcated into phases, each consisting 1 million instructions. Longer phase length is not used since intra-application variations in HSIP are not analyzed. As such, a conservative assumption is made that the HSIP changes very frequently. After each program phase, the runtime manager observes the observed time and averaged power values. The static estimate of expected TPI and wattage is used to calculate the expected execution time (and power consumption) after the application phase as well. The actual and expected values of the primary constraint are compared and the runtime manager chooses alternative configurations instead of the statically chosen configuration as explained in Section 6.2.3.1.

In our experiments, it has also been observed that the baseline POL (and WOL) order is not valid for all the application phases. On an average, the baseline performance order is invalid for about 95% of all application phases considered (toted for all benchmarks) while the baseline power order is invalid for about 50% application phases. Our two dynamic adaptation strategies differ in their management of POL and WOL at runtime.

In the first strategy referred to as the dynamic non-learning (DNL) adaptation, the POL and WOL are unaltered during execution. In the second strategy referred to as the dynamic learning based adaptation (DL), the POL (or WOL) are constantly updated if the existing order is found to be violated during execution. To understand the working of the learning based dynamic adaptation strategy, consider the following example when performance is the
primary constraint. Suppose it is found that positive slack manifests after phase $i$, and the corresponding configuration used for phase $i$ was $C_k$. The runtime manager chooses a slower configuration $C_l$ as per the POL for phase $i + 1$ to reduce power consumption. If the time consumed for phase $i + 1$ is found to be lesser than the time consumed for phase $i$, the position of $C_k$ and $C_l$ in the POL are interchanged. Such interchanges help the runtime manager track the varying requirements of the different application phases over time. This strategy has a small pitfall. When a configuration providing low performance on an average ends up at the top of the POL, it will hinder the runtime manager from properly managing negative slack in the future. Such a situation can arise if this configuration performs well for a few intermediate phases and is thus promoted to the top of the POL. To avoid this, the ends of the POL and WOL are periodically reverted to the corresponding configurations at the ends of the baseline ordered lists.

6.4 Evaluation

6.4.1 Evaluation Methodology

A commonly used set of benchmarks from the SPEC 2006 suite have been utilized for evaluating the newly developed architectural adaptation strategies. Multiple $U_d$s are synthetically generated corresponding to different operating modes described in Chapter 4. Since the objective of the current research is the satisfaction of $U_d$s, it is chosen to measure the percentage inaccuracy $PI$ associated with the satisfaction of both performance ($p\_in$) and power ($w\_in$) constraints individually. For each constraint, the corresponding $PI$ becomes 0 if it is satisfied. Otherwise, $PI$ is calculated as

$$p\_in = \frac{(P_d - P_{act}) \times 100}{(6.8)}$$

$$w\_in = \frac{(W_{act} - W_d) \times 100}{(6.9)}$$

where $P_{act}$ and $W_{act}$ are the delivered performance and power consumption respectively.

Figure 6.2 shows our evaluation platform. The $PWC$ corresponding to the different application phases observed when using all the allowed configurations are first collected through interval simulations using Sniper simulator. These values can be obtained using hardware coun-
ters provided on the chip. Since such an adaptive hardware is not available, values obtained through simulations are used. Synthetic variations are also generated in the schedule to model profiling inaccuracy and runtime effects. The system behavior with and without variations is separately investigated.

The Metadata database is populated according to the steps described in Section 6.2.2.3. At the end of every application phase \( i \) except the last, the runtime manager fetches the expected values of performance \( (P_{\text{expect}}) \) and power \( (W_{\text{expect}}) \) from the Metadata database. Similarly, the configuration selected by the SCG for the next phase \( (C_{i+1}) \) and the POL (or WOL) for phase \( i + 1 \) is fetched from Metadata database. A deviator module is responsible for generating actual PWC. This module takes as input the expected PWC values at the end of phases \( i \) and \( i - 1 \), and a special dirn variable as input. A deviation \( \text{dev} \) is applied to the execution time and power consumed while utilizing \( C_i \) during phase \( i \) in this process. \( \text{dev} \) is randomly generated between 0 and \( \text{dev}_{\text{max}} \) (set to 20% for our experiments). Larger values for \( \text{dev}_{\text{max}} \) are not considered at this point. The dirn variable, which stays constant for all application phases, can be given one of three values 0, 1, or -1. If \( \text{dirn} \) is set to 0, no deviations are produced. Setting the dirn variable to 1 (-1) produces negative deviation to power (performance) while producing positive deviation to performance (power) subject to the bounds explained earlier.

For the static cum dynamic adaptation strategy, the runtime manager selectively modifies \( C_{i+1} \) in the presence of slack affecting primary constraint. A configuration reached by moving Step size entries away from the index of original \( C_{i+1} \) in the corresponding ordered list (POL

![Figure 6.2 Evaluation platform](image-url)
or WOL) is chosen. This process repeats for all the application phases till the end of execution. At the end of execution, $p_{in}$ and $w_{in}$ are calculated.

For the dynamic adaptation strategy without learning, the runtime manager considers a single configuration selected statically for adjustment in all application phases. A single POL or WOL predetermined statically is also considered. For each phase, the runtime manager adjusts the baseline configuration similar to the static cum dynamic adaptation strategy. For the dynamic adaptation strategy with learning, the POL and WOL are also modified additionally based upon runtime observations as explained in Section 6.3.

6.4.2 Determination of maximum interval length

As mentioned earlier, smaller application phase length leads to a larger number of application phases. This allows fine-grained adaptation decisions, leading to better demand satisfaction. Simultaneously, the net adaptation overhead increases. In Section 6.2.1, the maximum interval length has been set to 10 million instructions. Further observations made in terms of the decrease in number of phases and degradation in $PI$ for demand satisfaction as maximum phase length increases have been used as the basis for this judgment. The decrease in the actual number of program phases as the maximum interval length is increased from 1 million instructions to 25 million instructions is first noted down. Figure 6.3 shows how the phase count scales as the maximum interval length is varied. In the figure, the maximum interval lengths investigated are shown on the x-axis. The values along y-axis are the maximum, minimum, and average (over different benchmarks) phase counts as a percentage of phases demarcated when constant interval length of 1 million instructions is employed. It can be seen that the number of phases drop very quickly as the maximum interval length is increased to 10 million instructions. After this point, the drop saturates slowly. For two benchmarks, namely hammer and specrand, the phase count scales down almost linearly with the maximum interval length. This shows that the performance and power profiles stay fairly constant throughout the execution for these benchmarks.

The adaptation approach specified in Section 6.2.2 to select a configuration per individual phase demarcated for the scenarios with different maximum interval lengths. Most aggressive
demands from balanced and stringent modes are input and the resulting PI values are measured. For each mode, 10,000 demands are generated and the reported PI values are averaged over these demands. Figure 6.4 shows the observed PI values. In the figure, the x-axis denotes the maximum allowed interval length. The y-axis shows both the average and maximum degradation in PI when compared to the PI observed for a constant interval length of 1 million instructions. As expected, the PI degrades as the maximum interval length is increased. The PI degradation is guaranteed to stay below 5% when the maximum interval length is restricted to a value below or equal to 10 million instructions. Following these trends, the maximum interval length is set at 10 million instructions.
6.4.3 Adaptation strategies

6.4.3.1 Benefit of intra-application adaptation

Static intra-application adaptation (referred to as SC) exploits the runtime variation in hardware-software interactions. Such an adaptation strategy can lead to better tradeoff decisions compared to an adaptation strategy that employs a single configuration for the entire application based on the inherent algorithm behavior (strategy referred to as ONE). As the configuration space is pruned, the benefit of intra-application adaptation decreases since the available performance-power points are discrete in nature. To evaluate the merit of the pruned configuration space in enhancing tradeoff through intra-application adaptation, the PI observed for the different demand scenarios is measured. These values are reported in Figure 6.5. In the figure, the x-axis denotes the different demand scenarios and the y-axis denotes $p_{in}$ and $w_{in}$ separately. It is observed that the primary constraint is always satisfied except for the low power demands. The considered configuration space cannot serve some of the low power constraints, due to which a small amount of $w_{in}$ is observed. As expected, the inaccuracy in serving user demands increases as $Pd - Wd$ increases. The PI values are highest for stringent demands since in this region of operating spectrum, high performance requires high power consumption. Intra-application adaptation saves 10% additional power when serving stringent demands when $Pd - Wd = 0.3$. For balanced intermediate demands on performance and power, the observed power saving due to intra-application adaptation is 5%.
6.4.4 Handling runtime variations in performance and power

Static adaptation strategy results in optimal tradeoff subject to the correctness of expected performance and power consumption knowledge. However, such expectations typically do not hold in a real-world scenario. Variations can arise due to runtime effects like cache contention, temperature induced mobility degradation, operating system management, inaccuracy of performance and power profiling, etc. When real execution proceeds faster than expected, the available performance slack should be utilized to reduce inaccuracy in tracking power demands. Similarly, situations with lower actual performance need to be handled by selecting alternate aggressive configurations to provide the required performance. The runtime manager designed for static cum dynamic adaptation strategy (referred to as SDC) is meant to handle such variations. The evaluation procedure described above is used to measure the effectiveness of the runtime manager in handling runtime variations. In Figure 6.6, the $PI$ values observed for $SC$ and $SDC$ strategies when (a) $dirn=1$ and (b) $dirn=-1$ respectively are reported.

When $dirn$ is set to 1, the available positive performance slack is traded off to reduce the power consumption. This effect is more pronounced for stringent demands which otherwise result in high $w_{in}$. In particular, it is observed that a 6% decrease in $w_{in}$ occurs when $P_d - W_d = 0.3$. Due to the discrete nature of the available performance-power points, the runtime manager overdid this tradeoff. This resulted in a slight increase in $p_{in}$. However, this effect is found to be negligible. When serving low power demands, performance is aggressively traded off to constrain power which is the primary constraint. This manifested as a 1.2% increase in $p_{in}$.

When $dirn$ is set to -1, power consumption is aggressively traded off to satisfy performance constraint for all except low power demands. This results in $\sim 5\%$ lower $p_{in}$ while employing dynamic configuration adjustments. An increase in $w_{in}$ is also noticed. The $PI$ increases for stringent and balanced demands as $P_d - W_d$ increases and is observed as 7% when $P_d - W_d = 0.3$. From these observations, it could be concluded that the runtime manager helps track the primary constraint well when it is negatively affected by runtime deviations.
6.4.5 Comparison of $SDC$ and dynamic adaptation strategies

The $PI$ values observed for $SDC$ strategy depict the tradeoff that can be achieved when detailed $PWC$ for different application phases are readily available. Such expectations are possible in real-time system environments but are not entirely practical for general purpose computing. Our dynamic adaptation strategy uses just a baseline performance and power order and the measured $TPI$ and power values to perform intra-application adaptation. Figure 6.7 compares the dynamic only strategies with $SDC$ in terms of the observed $PI$ values.

The dynamic adaptation strategies are further compared with the static cum dynamic adaptation strategy. Dynamic adaptation strategy considers re-adaptation more frequently (every 1M instructions). However, they do not exercise the knowledge of $PWC$ for future application phases and are thus suboptimal. Figure 6.7 compares the $PI$ values obtained for these different strategies when the $dirn$ variable is set to (a) 0, (b) 1, and (c) -1. The x-axis in the figure denotes the different user demand scenarios considered. The observed $PI$ values for the performance and power constraints are separately jotted along the y-axis.
Figure 6.7 Comparison of PI for SDC and dynamic adaptation strategies
When runtime deviations do not occur, the SDC strategy provides the best tradeoff. Both learning based and non-learning dynamic adaptation strategies have their own merits. Learning based dynamic adaptation slightly lowers the PI for secondary constraint while increasing the same for the primary constraint. It is observed that suboptimal configurations, which are fast only for a few application phases but provide mediocre overall performance can sometimes settle at the top of POL. When such a situation arises, the corresponding configuration is repeatedly selected for periods of negative slack leading to lazy slack reclaim. Since such a configuration consumes lower power than the most aggressive configuration, a power saving is noted. This effect is pronounced for balanced demands when \( P_d - W_d = 0.2 \). In this case, 2% performance is sacrificed to reduce power consumption by 5%.

Since applications execute in phases, it is reasonable to expect that learning based adaptation strategy provides better system behavior when compared to the non-learning strategy. However, the observed PI values do not reflect this understanding. It is noticed that the POL and WOL change very rapidly and the learning based strategy is not provided with enough time to adapt to this change. In particular, it is observed that the POL changes between adjacent phases with a probability of 0.8.

When positive (performance) slack arises, both the dynamic strategies utilize it to reduce power consumption. The only exception in this regard is observed for the learning based dynamic strategy when serving balanced performance demands. As mentioned earlier, mediocre performance configurations are sometimes pushed to the top of POL. Since there exists positive slack, such configurations are not employed, and configurations in the middle of the POL are usually utilized. Note that a few aggressive configurations are pushed to this area of the POL. Since they are deployed, the overall power consumption increases. Performance constraints are always satisfied as well.

When negative slack arises, all three considered adaptation strategies sacrifice power consumption while trying to improve performance. The performance constraints are still not satisfied for high performance and stringent demands. However, the PI for performance stays low (\( \sim 3\% \)). The associated increase in power consumption is negligible. For all the considered slack scenarios, the dynamic adaptation strategies perform closely to the SDC strategy.
This means that it is possible to avoid the large overhead associated with SDC strategy without sacrificing the tradeoff significantly. Learning based dynamic adaptation strategy can be used when a slight reduction in performance is acceptable to lower power consumption. Alternatively, the non-learning strategy can be used when the baseline POL or WOL is easily available.

6.4.6 Scaling of power consumption with performance

Figure 6.8 shows the delivered performance when the performance demanded from the microprocessor is varied from 95% to 65% in steps of 5%. The different performance demands are plotted along x-axis. The y-axis in this figure denotes the obtained performance as a percentage of maximal value. The SC and ONE strategies always deliver at least the required performance. In particular, the ONE strategy overcompensates for performance since the available performance-power points are discrete and no re-adaptation is considered. SC utilizes the knowledge of hardware software interactions for all application phases, and thus provides almost exactly the required performance. Both the non-learning and learning based dynamic strategies provide slightly lower performance than required. This effect is pronounced when the required performance is high (95%). In this case, these schemes provide 2.5% and 4% lower performance respectively.

The resulting power consumption for these performance demands is compared as well. The observed power consumption values for the different adaptation schemes are depicted
in Figure 6.9. The y-axis denotes the power consumption as a percentage of the maximal value. The SC strategy results in the lowest power consumption while the ONE strategy results in the highest power consumption. Both the learning based and non-learning based adaptation strategies consume similar power. The former strategy results in slightly lower power consumption than the latter when performance demands are high. Similarly, the latter strategy results in slightly lower power consumption when performance demands are low. Both the dynamic adaptation strategies consume power within 5% range of the power consumed by SC strategy. The difference between these is higher when performance demands are between 80% and 90%.

6.4.7 Comparison with previous schemes

Kontorinis et. al. [61] proposed adaptation strategies based on a table-driven adaptive core to reduce peak power. The authors adjust a set of 10 different adaptive components to obtain good performance when the peak power is restricted to different levels. Since the configuration space is pruned aggressively, it is essential to analyze how the tradeoff is affected due to the pruning process. The performance obtained as a fraction of performance obtained for $S_{\text{max}}$ for both our schemes and the schemes developed in [61] are compared. The peak power is constrained at 70%. This bound is used for power consumption since it is the lowest power bound considered in [61] and the authors present the relevant observations as well. The configuration space considered in [61] for this peak power constraint consists of 132 configurations (against
that are considered for this research). Figure 6.10 shows the performance values (%). In the figure, x-axis denotes the different adaptation strategies while the y-axis shows the performance values. The first five strategies are developed by the authors in [61] while the last three are ours. It is observed that our \textit{SDC} strategy results in the highest performance. In cases where \textit{SDC} is not practical, our dynamic adaptation strategies provide 7% less performance when compared to the best strategies proposed in the considered previous research. We believe this degradation is acceptable considering the reduction in hardware complexity required for adaptation. Also, the authors of the table driven adaptation scheme report their performance values based upon execution characteristics of SPEC 2000 benchmarks. On the contrast, this research uses SPEC 2006 benchmarks for evaluation, which pose larger hardware requirements for provision of good performance. Thus it is expected that the performance gap between these strategies will diminish even further in reality.

The proposed dynamic adaptation strategies are also analyzed in terms of how well they lead to energy efficient execution. The energy/performance efficiency provided when different performance constraints are imposed are measured. In [31], the authors report that their dynamic machine learning based adaptation strategy leads to 74% energy efficiency. The reported value is normalized to an oracular ideal scheme that chooses the best configuration for each application phase. Since the configuration space is pruned to retain only the best possible configurations, it is expected that the normalized energy efficiency to go up. Figure 6.11 shows the normalized energy efficiency for our dynamic adaptation strategies. It is observed that
both the learning and non-learning dynamic strategies result in about 95% normalized energy efficiency when performance demand is 95%. This shows how well our configuration space pruning methodology results in selection of efficient configurations. The normalized energy efficiency decreases as the performance demanded decreases since the optimal oracular scheme gets to consider larger number of configuration combinations. However, the normalized energy efficiency never drops below 75% when performance demand is greater than 65%.

6.5 Conclusion

In this chapter, the details of a two-stage adaptation strategy for microarchitectural adaptation are first presented. The first stage gathers and utilizes comprehensive information regarding expected $PWC$ for the entire application to statically determine a set of configurations to be used for the various application phases. A lightweight runtime manager is designed to account for the differences between expected and actual $PWC$ as part of the second stage. Two alternate dynamic only adaptation strategies are developed for situations where static profiling for fine-grained gathering of application wide $PWC$ is impractical. These strategies can easily be adopted even in multicores by specifying core level $TPI$ and wattage requirements.

An extensive evaluation process is employed to analyze how the newly developed adaptation strategies cater to widely variant demands from hardware. In particular, it is noticed that primary demands can be always be served with less than 2% inaccuracy on an average. The inaccuracy in serving the secondary constraint never rises above 10% unless very low power
consumption is not demanded along with ultra-high performance. The power required to serve 95% and 90% performance demands are close to 80% and 75% respectively. The performance provided by the dynamic adaptation strategies while constraining peak power to 70% is about 7% lower compared to schemes utilizing 10 times the configuration space. Our dynamic adaptation strategies also lead to energy efficient execution where the efficiency normalized to an ideal oracular scheme is about 95% when a similar performance is demanded.

All in all, the major contribution of the research details presented in this chapter lies in making microarchitectural adaptation more tractable. The issues of when to adapt and how to adapt are tackled. The corresponding techniques developed will make microarchitectural adaptivity an elegant solution for performance-power tradeoff and lets the user dictate the hardware behavior in a simplistic and flexible manner.
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this dissertation, we address the issue of using microprocessor systems in an effective manner to achieve a balance between performance, power, and reliability. In particular, we address the challenges of avoiding thermal cycling at the core level to minimize the reliability threat. We also address the challenge of providing maximum performance for a set power constraint, or the consuming minimum power for a given performance constraint.

Provision of good performance is at odds with lowering power consumption and improving reliability. Higher performance requires the inclusion of a larger number of transistors, thereby increasing power consumption, and possibly the chip temperature. The increase in and runtime fluctuations in chip temperature further affect reliability. As these factors are interrelated, it is not sufficient to consider the optimization of a single entity among them. In the current research work, we provide mechanisms to co-manage performance and reliability, and performance and power.

Design of aggressive cores for good single-threaded performance, as well as the aggregation of a large number of such cores to suit current software needs lead to chip reliability concerns. Factors affecting chip reliability include the chip temperature as well as its fluctuations over time. The latter factor, which is otherwise referred to as thermal cycling, has been identified as a major concern in previous research. However, schemes targeting reduction of thermal cycling have not been investigated. In this research, we provide mechanisms to keep both the chip peak temperature and the temperature fluctuations in check while adhering to set performance constraints. A real-time task execution environment with Quality of Service (QoS) guarantees is considered to enforce performance demands. The capabilities of DVFS and microarchitectural
adaptation are leveraged to select appropriate hardware configurations for the execution of tasks in a set schedule.

A two stage configuration selection scheme is developed to co-manage performance and reliability. The first stage statically selects the hardware configurations for individual tasks based upon knowledge of configuration-wise timing and temperature characteristics. Two alternate algorithms for configuration selection, namely *peak reduction* and *window based selection*, are developed and their effectiveness is analyzed. It is found that the former algorithm is slower but results in lower thermal gradients when compared with the latter algorithm. The second stage of configuration selection selectively alters the statically chosen configurations for deployment based upon runtime slack conditions. We have observed a 3-48 fold increase in chip lifetime expectancy pertaining to several failure mechanisms when our configuration selection schemes are employed on a schedule with 8 tasks. The increase in lifetime expectancy pertaining to thermal cycling is about 20 fold.

A major hindrance to the employment of microarchitectural adaptation is the control complexity. The size of possible configuration space prohibits the necessary analysis to decide the optimal configuration based on task execution characteristics. Previous research provides certain cues on how to reduce the configuration space. However, no formal approach to configuration space pruning exists. We designed a three stage methodology to bring down the configuration space to any desired size. The pruning methodology is based on application-specific expected performance and power characteristics made available through interval simulations. Multiple mechanisms to prune the configuration space are developed and compared. Our observations indicate that the pruned configuration space can be used to provide varied performance and power consumption levels with up to 92% accuracy. Since only the most useful configurations are retained after pruning, the presence of a fault can potentially degrade the system behavior. Our analysis in this regard shows that this degradation can be masked by up to 95% by utilizing the still available configuration space.

We further investigate mechanisms to provide performance-power tradeoff with the pruned configuration space exploiting program phases. A two stage comprehensive static cum dynamic adaptation strategy is developed that exploits phase-wise knowledge of performance and power
characteristics. Similar to our performance-reliability co-management scheme, the static component strives to preserve tradeoff optimality while the dynamic component deals with runtime variations. A similar approach to adaptation is not possible without the configuration space pruning due to the complexity of the associated tradeoff optimization problem. We also develop two alternate dynamic adaptation strategies that provide the required tradeoff without the knowledge of phase-wise operating characteristics. It is observed that single constraint demands (on performance or power) are served with $\sim 98\%$ accuracy. For demands involving both performance and power, the inaccuracy in tracking the demands rarely crosses $10\%$. For a set power consumption level, our adaptation strategies provide $93\%$ of the performance provided by a previously proposed strategy that considers 10 times the configuration space size. Our dynamic adaptation schemes result in about $95\%$ of the maximum possible energy efficiency against $75\%$ possible for a related state-of-the-art scheme adapting 14 billion configurations.

To summarize, our current research addresses the various aspects associated with microprocessor performance-power and performance-reliability co-management. Various mechanisms to obtain required operating characteristics from hardware are proposed. These mechanisms leverage on \textit{DVFS} and microarchitectural adaptation. Previously unconsidered ill-effects of thermal cycling on chip reliability are included to provide holistic performance-reliability co-management solutions. The adaptive microarchitectural configuration space is reduced as per set requirements and lightweight schemes are developed for configuration selection. Our research demystifies the complexity involved in microarchitectural adaptation and motivates the design of such architectures.

\subsection{Future work}

Our research currently deals with uniprocessor performance-reliability and performance-power co-management. In the future, this will be extended to address similar issues in multicore. Our dynamic performance-power adaptation strategies are amenable usage in the multicore scenario. Per core \textit{CPI} and wattage requirements can be set up from the given performance and power budget/demand. Core level adaptation can be performed using the proposed strategies. The demands can further dictate the number of cores to utilize for a given
Our preliminary analysis in this regard confirms this claim. For example, figures 7.1 and 7.2 show the variation of normalized power with the normalized performance for FFT and Barnes hut method for solving N-body interactions. A multicore platform with 4 cores is used for the analysis. Either 1, 2, or 4 cores can be made available for application execution.

In these figures, the normalized performance is represented on x-axis. The y-axis shows the normalized power consumption. The performance and power consumption values are normalized with respect to the values obtained with execution on 1 processor with maximal configuration. It is observed that the benefit of using either 1, 2, or 4 cores can be associated with different regions of performance spectrum. Also, these regions are application dependent. As such, the performance demand and the application at hand can be first used to decide the number of cores to utilize. The performance and power budget can be decomposed to core-level budgets. Following this, the required tradeoff using the selected cores can be provided using our dynamic adaptation strategies. Our analysis in this regard is still preliminary and further experimentation is needed to formalize the tradeoff methodology.

We are also interested in building a hardware prototype for the envisaged adaptive processing platform. FPGAs provide an excellent platform for such hardware emulation. Current FPGAs come with an included general purpose processor integrated into the chip fabric. This processor can be used to make adaptation decisions which can then be communicated to an adaptive processor configured on the FPGA. A thorough analysis will be performed to under-
Figure 7.2 Normalized performance vs. normalized power for Barnes hut algorithm

stand and implement the required adaptivity in hardware.

The schemes proposed in this research can also be extended to include other components of computing platforms as well. Nest configuration, on-board graphic cards, and shared caches can be included to provide comprehensive system level performance, power, and reliability management mechanisms. Such mechanisms have a higher impact on the considered operating characteristics when compared to mechanisms considering individual cores.
Bibliography


[16] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. Low-power cmos

and assignment for hard real-time applications on mpsocs. *Very Large Scale Integration

[18] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on

[19] Wayne H Cheng and Bevan M Baas. Dynamic voltage and frequency scaling circuits with

[20] Bruce Childers, Hongliang Tang, and Rami Melhem. Adapting processor supply voltage

[21] Thomas F Coleman and Yuying Li. An interior trust region approach for nonlinear


agement in multiprocessor socs using online learning. In *Design Automation Conference,


[49] Intel. First the tick, now the tock: Next generation intel microarchitecture (nehalem), 2008.


[89] Manish Shah, J Barren, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, et al. Ultrasparc t2:


