ADAPTIVE THREAD MANAGEMENT FOR POWER, TEMPERATURE, AND RELIABILITY IN FUTURE MICROPROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jonathan Aaron Winter

February 2010

ADAPTIVE THREAD MANAGEMENT FOR POWER, TEMPERATURE, AND RELIABILITY IN FUTURE MICROPROCESSORS

Jonathan Aaron Winter, Ph. D.

Cornell University 2010

With continued scaling of CMOS technology, power, thermal, and reliability issues threaten to significantly limit future performance improvements. The advent of microprocessors with multiple processing units creates a new opportunity to address these concerns through low-cost adaptive thread management techniques. In this dissertation we devise two types of dynamic management schemes, thread migration and power management, which leverage the inherent architectural characteristics of future microprocessors to dramatically mitigate thermal hotspots, variations, and hard faults. These techniques are applied both within the core, in clustered simultaneous multithreaded (SMT) architectures, and among the cores of unpredictably heterogeneous chip multiprocessors (CMPs).

First, we investigate dynamic thermal management (DTM) in clustered SMT architectures. We propose novel thread migration algorithms that leverage the steering mechanism inherent in clustered architectures to cool hotspots more effectively than dynamic voltage and frequency scaling (DVFS) when executing thermally non-uniform workloads. In addition, we create a DTM mechanism that combines intelligent steering with DVFS power management to achieve efficient thermal control across all workloads.

In future large-scale multi-core microprocessors, hard faults and process variations will create dynamic heterogeneity, causing performance and power

characteristics to differ among the cores in an unanticipated manner. Contemporary CMP thread managers are oblivious to this heterogeneity, resulting in significant performance losses and excess power dissipation. We develop operation system scheduling and global power management policies, which significantly reduce the loss in power-performance efficiency. We further explore the scalability of these algorithms to many-core architectures with four to two-hundred fifty-six cores and devise novel, scalable runtime management techniques which achieve high performance with low overhead.

BIOGRAPHICAL SKETCH

Jonathan Aaron Winter graduated with honours from the University of Toronto with a Bachelor of Applied Science in Computer Engineering. During his three summers between undergraduate years, Jonathan applied his engineering course work on practical problems in academic research and industry. In the summer of 2000, Jonathan worked as a summer research student for Professor Tarek Abdelrahman in the Department of Electrical and Computer Engineering on data structure analysis and visualization. The following summer he worked as summer intern in Software Engineering at Altera Corporation's Toronto Technology Centre. Finally, in the summer of 2002, Jonathan was awarded an NSERC Undergraduate Student Research Award to work with Professor Fahiem Bacchus in the Department of Computer Science at the University of Toronto. That summer he conducted research on preprocessing algorithms for Boolean Satisfiability which was presented at the International Conference on Theory and Applications of Satisfiability Testing.

In August 2003, Jonathan enrolled in a M.S./Ph.D. program at Cornell University in the Department of Computer Science. In his first year and half, Jonathan passed the Computer Science Department's qualifying exams and conducted research at IBM's T.J. Watson Research Center. Jonathan began working under the supervision of Professor David H. Albonesi in January 2005 on his Master's and Ph.D. research. Together, Jonathan and Professor Albonesi published an article in ACM Transactions on Architecture and Code Optimization, and presented papers at the International Conference on Dependable Systems and Networks and the Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures all in 2008. Jonathan received his M.S. degree in August 2008 and defended his Ph.D. in August 2009.

In September 2009, Jonathan joined Google Inc. as a Software Engineer, to work in the Platforms Group on software for datacenter power management.

This dissertation is dedicated to my Mom and Dad for all their love, support, advice, and *patience* throughout the years.

ACKNOWLEDGMENTS

I would first like to acknowledge the endless advice and support of my advisor Professor David Albonesi who with a steady hand and great wisdom, crafted me into the researcher I am today. Dave's efforts were critical to my success, including his insistence on working *at least* 60 hours a week, his timely suggestions on research directions, his tireless editing and reediting of my papers and presentations, and his enduring faith during the tough times of paper rejections and career uncertainty. Finally, I can say that all those 60 hour work weeks paid off!

I want to show my appreciation to the three other members of my committee, Rajit Manohar, Bart Selman, and Christine Shoemaker for their feedback, critique, and research guidance. I would also like to acknowledge the support and collaboration of the "Albonesi Research Group" – Matt Watkins, Paula Petrica, Basit Riaz Sheikh, and Mark Cianchetti. Hopefully, my success brings hope that there *is* a light at the end of the tunnel. In addition, I want to thank all the member of the Computer Systems Laboratory (CSL!) for providing a sounding board for ideas, getting me out of the office once in a while, putting up with my bad sense of humour, and sitting through so many of my talks. Carry the computer architecture (and async) torch high and far.

One other Cornellian deserves acknowledgment for being fundamental to my success. I want to thank Dr. Lindsey Banigan for her endless encouragement throughout the last three and a half years. I honestly don't think I could have finished my Ph.D. without you, and my fondest memories of graduate school were with you.

Finally, I would like to thank Michael, Debbie, Dad, and Mom for inspiring me to be the best I could be and putting up with me for trying. My brother is clearly the smartest of the three of us for not even contemplating doing a Ph.D. My sister probably figures that no Ph.D. could be as bad as growing up with us for two older brothers. Seriously, your love and support has made this dissertation possible.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vi
List of Figures viii
List of Tables x
Chapter 1: Motivation and Research Outline 1
Chapter 2: Related Work 8
2.1: Power-Efficient Computing 8
2.2: Dynamic Thermal Management 11
2.3: Mitigating Hard Errors and Variations 17
2.4: Designed-Heterogeneous Chip Multiprocessors 21
2.5: Early Work on Unpredictably Heterogeneous Chip Multiprocessors 22
Chapter 3: Addressing Thermal Non-Uniformity in SMT Workloads 25
3.1: Introduction 25
3.2: The Clustered SMT Microarchitecture 29
3.3: Methodology 34
3.4: Steering-Based Dynamic Thermal Management Policies 39
3.4.1: Dispatch Gating Policies 40
3.4.1.1: Results 42
3.4.2: Heat Spreading Policies 43
3.4.2.1: Static Heat Spreading Policies 44
3.4.2.2: Static Heat Spreading Results 45
3.4.2.3: Dynamic Heat Spreading Policies 46
3.4.2.4: Dynamic Heat Spreading Results 50
3.5: Combined Steering and DVFS Policies 54
3.6: Conclusions 56
Chapter 4: Application Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures 58
4.1: Introduction 58
4.2: Scheduling Algorithms for Unpredictably Heterogeneous CMPs 60
4.2.1: Baseline Scheduling Algorithms 61
4.2.2: Hungarian Scheduling Algorithm 62
4.2.3: Iterative Optimization Search Algorithms 64
4.3: Methodology 67
4.4: Results and Discussion 73
4.4.1: Baseline Scheduling Algorithms 73
4.4.2: Hungarian Scheduling and Search Algorithms 74
4.4.3: Overall Comparison 76
4.5: Conclusions 77
Chapter 5: Global Power Management Algorithms for Unpredictably Heterogeneous CMP Architectures 79
5.1: Introduction 79
5.2: Global Power Management 80

5.2.1: Baseline DVFS Algorithms 81
5.2.2: Throughput-Aware Power Allocation Scheme 92
5.2.3: Up/Down DVFS 97
5.3: Methodology 101
5.4: Results and Discussion 103
5.5: Possible Extensions 107
5.6: Conclusions 108
Chapter 6: A Scalability Analysis of Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures 109
6.1: Introduction 109
6.2: Methodology 110
6.3: Results and Discussion 112
6.3.1: Scheduling Algorithm Performance 112
6.3.2: The Impact of the Number of Intervals on the Search Algorithms 116
6.4: Conclusions 117
Chapter 7: Scalable Thread Scheduling and Global Power Management for Future Unreliable Many-Core Architectures 119
7.1: Introduction 119
7.2: Adaptive Thread Management for Unreliable Many-Core Architectures 120
7.2.1: Scalability Issues for Adaptive Thread Management Algorithms 122
7.2.2: Coordinating Application Scheduling and Global Power Management 124
7.3: Application Scheduling and Global Power Management Algorithms 126
7.3.1: Overview 126
7.3.2: Application Scheduling Algorithms 127
7.3.3: Global Power Management Algorithms 143
7.4: Methodology 157
7.4.1: Simulation Infrastructure 157
7.4.2: Simulating Unpredictably Heterogeneous Many-Core Processors 158
7.4.3: Workloads 159
7.4.4: Assessing Algorithm Runtimes 160
7.5: Results and Discussion 160
7.5.1: Coordinating Application Scheduling and Global Power Management 160
7.5.2: Application Scheduling Algorithms 164
7.5.3: Global Power Management Algorithms 168
7.5.4: Online Results 170
7.6: Conclusions 172
Chapter 8: Opportunities for Future Work 174
Chapter 9: Conclusions 178
References 180

LIST OF FIGURES

Figure 3.1: vortex / equake – performance with no DTM mechanism …… 26
Figure 3.2: vortex / equake – performance with dynamic voltage and frequency scaling ……27
Figure 3.3: vortex / equake – temperature with dynamic voltage and frequency scaling ……28
Figure 3.4: A clustered simultaneous multithreaded microarchitecture……... 30
Figure 3.5: The floor plan of the clustered SMT microarchitecture…… 32
Figure 3.6: The three dispatch gating techniques……..42
Figure 3.7: Performance of the dispatch gating policies on the clustered SMT design ……43
Figure 3.8: The two static heat spreading techniques……... 45
Figure 3.9: Performance of the static heat spreading policies on the clustered SMT design ……46
Figure 3.10: The four dynamic heat spreading techniques …… 47
Figure 3.11: Performance of the dynamic heat spreading policies on the clustered SMT design ……51
Figure 3.12: vortex / equake – performance with the Counter-Based Steering policy ……. 52
Figure 3.13: vortex / equake – temperature with the Counter-Based Steering policy ……. 53
Figure 3.14: Overall performance comparison of the DTM techniques on the clustered SMT design……. 55
Figure 4.1: An illustrative example of an eight-core unpredictably heterogeneous chip multiprocessor ……59
Figure 4.2: An example of the two-phase scheduling approach …….. 61
Figure 4.3: The Round Robin Scheduling Algorithm ……..62
Figure 4.4: The Randomized Scheduling Algorithm ……... 62
Figure 4.5: Munkres' six step Hungarian Algorithm ……64
Figure 4.6: The Global Search Algorithm ……65
Figure 4.7: The one swap Local Search Algorithm …… 66
Figure 4.8: The two swap Local Search Algorithm …… 66
Figure 4.9: The hierarchical and parallel multi-core simulation framework …….... 68
Figure 4.10: Processor core floor plan ……... 70
Figure 4.11: Comparison of the baseline scheduling algorithms ……...73
Figure 4.12: Comparison of advanced scheduling algorithms ……...75
Figure 4.13: Overall comparison ……77
Figure 5.1: Global power management definitions ……..82
Figure 5.2: Assumptions in GPM power-performance modeling ……...83
Figure 5.3: Formulas for estimating power and throughput using model ……84
Figure 5.4: The traditional formulation of global power management as an optimization problem ……84
Figure 5.5: An overview of global power management using the MaxBIPS algorithm ……88

Figure 5.6: An overview of global power management using the TAPAS algorithm ……93
Figure 5.7: The TAPAS formulation of the global power management optimization problem ……..94
Figure 5.8: Additional definitions for Up/Down DVFS …….. 98
Figure 5.9: Throughput improvement for the global power management schemes across the degraded configurations ……..104
Figure 5.10: Throughput improvement for the global power management schemes across the benchmark workloads …….105
Figure 5.11: Percent performance lost relative to no GPM across the degraded configurations …….. 106
Figure 5.12: Percent performance lost relative to no GPM across the benchmark workloads …….107
Figure 6.1: The power-performance efficiency results for the scheduling algorithm scalability study ……...113
Figure 6.2: The impact of the number of intervals on the effectiveness of the search algorithms …….117
Figure 7.1: Scheduling and power management time quanta and sampling periods ……..121
Figure 7.2: The growth in the runtime of various algorithm complexity classes ...124
Figure 7.3: Formulas used in the adaptive thread management algorithms ……...127
Figure 7.4: Definitions for adaptive thread management algorithms …….128
Figure 7.5: Definitions for the Hierarchical Hungarian Scheduling Algorithm … 138
Figure 7.6: The formulation of global power management as a linear program …155
Figure 7.7: Linear approximation methods for the voltage/power relationship ….156
Figure 7.8: Performance loss of uncoordinated scheduling and global power management algorithms relative to the oracle manager …….. 161
Figure 7.9: Power dissipation of uncoordinated scheduling and global power management algorithms and the oracle manager …….162
Figure 7.10: A study of the impact of scheduling and power management on many-core processors ……..... 164
Figure 7.11: Scheduling algorithm runtimes as a percentage of the scheduling quantum ……... 165
Figure 7.12: The impact of group size on runtime overhead for the Hierarchical Hungarian Algorithm ……...166
Figure 7.13: The impact of group size on performance loss for the Hierarchical Hungarian Algorithm ……...167
Figure 7.14: Scheduling algorithm performance percentage loss relative to the Hungarian Algorithm ……...168
Figure 7.15: Global power management algorithm runtimes as a percentage of the power management quantum …….169
Figure 7.16: Power management algorithm performance percentage loss relative to the LinOpt algorithm ……... 170
Figure 7.17: Online performance results for application scheduling and power management algorithms with 256 core CMPs over 2 application scheduling quanta …… 171

LIST OF TABLES

Table 3.1: Simulated clustered SMT microarchitectural parameters ……...33
Table 3.2: Baseline benchmark set performance and thermal characteristics ……..38
Table 4.1: Scheduling algorithm summary …….. 61
Table 4.2: Core architectural parameters ……..... 69
Table 4.3: The degraded CMP configuration ……...71
Table 4.4: Application workloads …… 72
Table 5.1: Power levels for chip-wide DVFS, per-core MaxBIPS DVFS, and TAPAS ……85
Table 5.2: Core architectural parameters ……101
Table 5.3: List of faults and variations affecting each core in the 3 degraded …...102
Table 5.4: Application workloads …….. 103
Table 6.1: List of possible forms of core degradation ……112
Table 7.1: A summary of the features of the application scheduling algorithms ...143
Table 7.2: A summary of the features of the global power management algorithms ……..157
Table 7.3: Possible forms of core degradation ……...159

CHAPTER 1

MOTIVATION AND RESEARCH OUTLINE

While performance will always be a primary concern for computer architects, other design considerations have come to the forefront in the deep-submicron microelectronics technology era. First of all, power dissipation has become a major concern across product domains [21][92]. In embedded and portable electronics, high levels of power dissipation reduce battery life and increase package sizes. For the desktop market, high power increases production costs because larger, more expensive thermal packages are required. Finally, at the server level, high power requirements lead to increased datacenter operating expenses because of the added energy supply and cooling costs.

Thermal management is another critical design consideration for modern microprocessor design. Chip overheating can lead to timing errors, reduced reliability, and even immediate thermal damage to the die. To some extent, thermal concerns are connected to power dissipation because increased power leads to higher chip temperatures and thus techniques for reducing power also contribute to addressing overheating. While average die temperature is strongly correlated to overall power dissipation, peak chip temperature and thermal hotspots are a different yet important matter [126].

A final area of concern that has recently taken center stage in general-purpose computer architecture is hardware reliability. While transient (soft) errors are the primary focus today, permanent (hard) errors and circuit variability are expected to become a major challenge in the future [13][14]. Hard faults and variability cause problems during chip production as well as over the lifetime of the processor. Faults such as open or shorted wires or improperly manufactured transistors can cause errors which impede the function of part of the processor, potentially rendering the chip

unusable. Manufacturing process variations can also reduce chip yield by increasing the number of dies produced which operate outside of specification in terms of clock frequency or power dissipation. Some chips with variability will be salvageable but need to be clocked at lower speeds or run at higher voltages to meet specification, leading to poor performance and power efficiency.

Over the lifetime of the chip, faults can also develop as a result of component wear-out through a variety of mechanisms including electromigration, stress migration, time-dependent dielectric breakdown (TDDB), thermal cycling, and negative bias temperature instability (NBTI) [127][129]. Variations hasten lifetime wear-out, forcing users to replace chips before their expected end-of-life. Process variations can cause transistors and wires to be manufactured in a weakened state. For example, transistors could be produced with thinner than nominal gate oxide increasing their susceptibility to TDDB while thinner than nominal wires would be more likely to suffer from electromigration. Voltage and thermal variations also accelerate many of the wear-out mechanisms [127][129].

In this dissertation, adaptive thread management techniques are developed to tackle the challenges of power dissipation, thermal hotspots, and chip reliability, while simultaneously seeking to maximize processor performance. Two microprocessor design styles are studied, *clustered simultaneous multithreaded (SMT)* architectures and *chip multiprocessors (CMPs)*. Clustered SMT processors consist of a traditional SMT front end, capable of fetching and decoding multiple threads, and a partitioned back end where the execution resources – the issue queues, register files, and function units – are divided among multiple separate pipelines or "clusters". CMPs are comprised of multiple processing cores placed together on a single die. Both architectures were developed to run multiple applications simultaneously on a single processor while minimizing design complexity. A major theme of the proposed

architectural enhancements is that they exploit the inherent diversity in the behavior of the running applications to increase performance and reliability while reducing power and temperature at low cost and complexity. For both microprocessor design styles, we explore thread migration and power management policies. In clustered SMT architectures, these techniques are applied within a single processor core, while CMPs provide the opportunity to globally optimize application scheduling and power management across cores with low overhead.

In Chapter 3, the focus is on *dynamic thermal management (DTM)* in clustered SMT architectures. The objective is to utilize the non-uniformity in heating among the simultaneously running threads to alleviate die hotspots without employing more heavyweight cooling mechanisms. DTM is a microarchitectural approach to maintaining acceptable on-die temperatures while reducing packaging and cooling costs [22][54]. Rather than designing the chip's thermal solution (heat sink and fans) for the absolute worst case, a cheaper cooling package is provided and thermal sensors are used to detect situations when the chip temperature is approaching the permitted maximum level. To avoid a thermal violation, microarchitectural techniques are employed which reduce power by throttling chip resource usage. While prior research has investigated DTM techniques for single-threaded [22][38][51][54][80][81][82] [99][124][126], SMT [39][40][50][65][74][100], and CMP [39][41][47][74][100] [130] architectures, clustered SMT processors provide the opportunity to develop new types of low-cost thermal management techniques. Because threads running on the processor execute in separate back-end clusters, there are spatial variations in the heating of the back end. DTM methods based on the cluster steering mechanism can be used to migrate threads to eliminate hotspots with low cost. Moreover, combining steering-based thermal control with dynamic voltage and frequency scaling (DVFS) provides a comprehensive DTM policy.

Chapters 4 through 7 investigate adaptive thread management techniques to reduce the impact of hard errors and variations in chip multiprocessors. Future CMPs are likely to be designed as a collection of homogeneous processing cores connected by an interconnection fabric. Using homogeneous cores reduces the design complexity and verification costs. As semiconductor technology continues to scale to smaller transistor and wires, hard errors are becoming more common and variations are becoming more extreme. Prior research has studied resiliency methods for tolerating these faults and variability, allowing processors with unreliable components to remain operable [1][2][15][16][17][18][36][77][78][79][87][94][109][116][122][123][129][135]. However, the faults and variability will leave these chips in a degraded state where broken components are disabled, clock frequency is reduced, and power dissipation is above nominal. Potentially, these functional but degraded processors will still be unusable because of their poor performance and power efficiency. The random processes creating these faults and variations will likely impact each core in the CMP differently, creating dynamic heterogeneity among the processor's cores. These *unpredictably heterogeneous* chip multiprocessors (UH-CMPs) create a new, more challenging environment than prior studied homogeneous and designed-heterogeneous CMPs, and will require novel, intelligent runtime management schemes.

In Chapter 4, we show that current operating system scheduling algorithms will assign applications to the degraded cores in an ineffective manner, oblivious to this heterogeneity. Adaptive thread scheduling algorithms are then presented which match application resource requirements to core functionality, greatly reducing the impact of the hard errors and variations. Two types of algorithms are developed. The first, called the *Hungarian Scheduling Algorithm*, makes some simplifications about the nature of the assignment problem. Next, it runs the Hungarian Algorithm [25][93]

which calculates the optimal solution to the simplified problem and then it applies this solution to the original problem. The second approach is to use iterative search techniques from artificial intelligence which are known to be effective on complex optimization problems. Both algorithms are experimentally shown to be highly effective at maximizing power-performance efficiency in unpredictably heterogeneous CMPs.

As architects continue to integrate more cores on a die, it is no longer sufficient to simply address power at the core level. Instead, power management becomes a chip-wide optimization problem which must balance the needs of individual cores and applications for maximum overall performance. The goal is to maximize the overall throughput of the CMP while ensuring that the total power dissipation does not exceed a global power budget. Chapter 5 introduces a new way of looking at global power management, called the *Throughput-Aware Power Allocation Scheme (TAPAS)*, which is unique because it focuses on setting power levels directly, rather then via voltage and frequency control knobs. The dynamic heterogeneity of future unreliable multi-core processors makes this environment a challenging medium for modeling the performance and power dissipation of DVFS. Consequently, TAPAS' lower reliance on analytical models and reduced complexity makes it ideal for global power management on UH-CMPs.

As the number of cores on a chip increases in future technology generations, the problem of intelligently assigning applications to cores becomes increasingly complex. In a chip multiprocessor running *n* applications on *n* cores, there are n! possible scheduling permutations that can be chosen. Chapter 6 studies the impact of scaling up the number of cores from 4 to 64 on the energy efficiency of the scheduling algorithms proposed in Chapter 4. We also characterize the influence of the number of

search iterations on the effectiveness of our iterative optimization algorithms across the different CMP sizes.

Chapter 7 continues studying algorithm scalability but expands the focus to both application scheduling and global power management (GPM) techniques and shifts the optimization objective to maximizing total chip throughput. First, the need for coordination between the thread scheduler and power manager is investigated. Then, this chapter provides a comprehensive analysis of the scalability and performance of adaptive thread management schemes on unpredictably heterogeneous many-core architectures with up to 256 cores. Each algorithm's runtime overhead is assessed using formal complexity theory as well as experimentally evaluated. The ramifications of core scaling on the sampling accuracy of the techniques are also investigated. A highly scalable *Hierarchical Hungarian Algorithm* is developed which has dramatically reduces runtime overhead compared to the poorly scaling Hungarian Algorithm, but achieves performance comparable to an idealized zero overhead scheduler. Finally, a Steepest Descent global power manager is shown to have minimal runtime costs, even for 256 core processors, yet beat the performance of state-of-the-art GPM techniques based on linear programming.

Possible opportunities for future work are discussed in Chapter 8. These include a) conducting a scalability analysis on the TAPAS algorithm; b) extending our runtime management framework beyond scheduling and power management to incorporate Complexity-Adaptive Processing [3][4]; c) augmenting our simulation infrastructure to more accurately model inter-core effects, such as cache sharing, off-chip bandwidth, and the on-chip interconnection network, as well as modeling the impact of process variations and hard faults in these components; d) investigating adaptive thread management for multithreaded applications; and e) developing

degradation- and phase-aware runtime management algorithms that eliminate the need for time consuming sampling during the exploration phase.

In the next chapter, an overview is given of prior work that relates to these research projects. This includes previous work on power-efficient computing in SMT and CMP architectures, global power management in multi-core processors, dynamic thermal management, tolerating hard errors and variations, and heterogeneous chip multiprocessors.

CHAPTER 2

RELATED WORK

A number of different research areas relate to adaptive thread management for power efficiency, thermal control, and reliability in clustered SMT architectures and chip multiprocessors. The first section of this chapter looks at prior work on power-aware architectures, including efforts for SMT processors, energy-efficient CMPs, and multi-core global power management. The next section discusses research on dynamic thermal management, and again focuses on SMT and CMP systems, as well as clustered architectures which closely resemble the study in Chapter 3. The third section highlights different research efforts which tackle detecting, isolating, and tolerating hard errors as well as modeling and compensating for process variations. The fourth section discusses prior work on scheduling and power efficiency for *designed-heterogeneous* chip multiprocessors where the asymmetry in the cores is an architected feature. Finally, we discuss a few research efforts that have started to address unpredictably heterogeneous systems in parallel to our work.

2.1. Power-Efficient Computing

In the past fifteen years, power dissipation and power-aware computing have become critical design considerations for general purpose processors [21][92]. The development of Wattch, an architectural-level power model by Brooks et al. [23], helped spur numerous research efforts aimed at making microprocessors more power-efficient. Here, the focus is on recent work on power-aware simultaneous multithreaded machines and chip multiprocessors as they most resemble the architectures studied in this dissertation.

A few prior efforts address the energy efficiency of SMT processors. Seng et al. [118] examine the power efficiency of SMT machines and propose architectural enhancements to optimize energy usage. Li et al. [75] perform a design space

exploration to characterize the energy efficiency of SMT architectures and identify the root causes of multithreading's power advantage. Another study compares the energy efficiency of chip multiprocessors and simultaneous multithreaded machines and finds CMPs to be more compelling [115].

Other previous research uses the operating system to improve CMP power efficiency. Juang et al. [62] argue for coordinated formal control-theoretic methods to manage energy efficiency in multi-core systems. Li and Martínez [72] investigate heuristics that adaptively change the number of cores used and the chip voltage and frequency to optimize power-performance in parallel applications. DeVuyst et al. [37] study a CMP consisting of SMT cores and show that it can be desirable to schedule threads in an unbalanced manner on the cores in order to be most energy-efficient. Herbert and Marculescu compare the energy efficiency of per-core DVFS schemes to those where a CMP is partitioned into voltage/frequency islands containing groups of cores and show that the per-core power management is not always necessary [52]. Lastly, Miao et al. employ genetic algorithms to control DVFS to reduce energy consumption in CMPs [90].

Lee and Brooks [71] focus on processor core complexity and evaluate the power-performance tradeoffs of pipeline depth and width on SMT and CMP architectures. Ekman and Stenstrom [43] also look at core complexity, but study the power-performance tradeoff of issue-width verses the number of cores when running parallel applications. On the other hand, Li et al. [76] and Monchiero et al. [91] take a broader view and study how the overall design of chip multiprocessors is impacted by the key constraints of power and temperature.

Most related to our work in Chapters 5 and 7 on global power management in CMPs is the research by Isci et al. [58] which introduces the problem of trying to maximize total throughput under a chip-wide power constraint by dynamically tuning

DVFS to workload characteristics. These authors develop the popular MaxBIPS algorithm which serves as the baseline per-core DVFS scheme in Chapter 5. Sharkey et al. [119] extend this work by exploring algorithms based on both DVFS and fetch toggling, and by exploring a number of design tradeoffs including the granularity at which the power manager is called and local verses globally managed techniques. Bergamaschi et al. also conduct further work on MaxBIPS and compare its discrete implementation to using continuous power modes [10]. The authors then develop a non-linear programming solution to solve the continuous power mode problem, which is not practical to implement in hardware but serves to confirm the effectiveness of discrete MaxBIPS. Kim et al. develop and analyze on-chip voltage regulators to allow for fine-grained per-core DVFS [64]. Using an offline DVFS algorithm, they find significant benefits for running voltage and frequency scaling at finer temporal and spatial granularities. Both the work of Sharkey et al. [119] and Kim et al. [64] demonstrate that fine-grained DVFS algorithms can have significant performance benefits and this motivates our development of TAPAS in Chapter 5.

Sartori and Kumar [114] propose decentralized power management algorithms for homogeneous many-core architectures. Although they only evaluate their algorithms for systems with up to 16 cores, the algorithms are designed for larger systems. One technique uses per-core steepest descent where unlike our implementation in Chapter 7, each core searches its power states independently. They propose alternative approaches to DVFS such as setting cores to high and low power states at a coarse granularity and migrating benchmarks at a finer granularity to meet the power budget. In a similar vain, Rangan et al. [105], explore the use of scheduling on cores statically set to different voltage and frequency levels as an alternative power management approach to fine-grained DVFS.

Meng et al. [88] develop a framework for the combined optimization of multiple power management mechanisms and suggest employing a greedy search algorithm to quickly find a good setting for each power scheme. This greedy search method is actually Steepest Descent and in Chapter 7 we adapt this technique for scalable DVFS control in many-core processors. Bitirgen et al. also examine coordinated management, but apply a machine learning approach to allocate chip resources (one of which is power) to meet various performance objectives [11]. Finally, Wang et al. [137] propose a coordinated approach to global power and temperature management based on optimal control theory. They use multi-input-multi-output (MIMO) control strategies and model predictive control (MDC) theory which are effective, but require matrix-matrix multiplication and either matrix inversion or factorization. These matrix operations are all high-order polynomial time algorithms that scale poorly and thus we do not pursue such methods in Chapter 7 when we study runtime management algorithms for many-core architectures.

2.2. Dynamic Thermal Management

Previous research on DTM techniques can be categorized into a number of different themes. First of all, a few studies pioneered the need for dynamic thermal management and proposed early solutions in this area. In 2003, Skadron et al. [126] released HotSpot, an easy to use architectural-level temperature model which generated much research activity. Recent research addresses thermal issues on processors with multiple threads (either through SMT or CMP). Finally, a few papers analyze DTM on single-threaded clustered microarchitectures.

Huang et al. [54] were perhaps the first to propose the need for dynamic thermal management. They outlined a framework called Dynamic Energy Efficiency and Temperature Management (DEETM) which invokes the operating system to monitor chip thermal behavior and engage various techniques to combat high

temperatures. Brooks and Martonosi [22] further strengthen the case for dynamic thermal management and outline the basic components and methodology upon which all DTM schemes are based.

Dhodapkar et al. [38] propose TEM^{2}P^{2}EST, an early thermal model at the architectural level. Lim et al. [82] use the TEM^{2}P^{2}EST model to explore a form of activity migration that uses a second low-power in-order pipeline when the main out-of-order pipeline overheats. Heo et al. [51] examine activity migration in detail and try to determine which microprocessor components are best duplicated. They conclude that it is most critical to replicate the register files and execution units to get the best power/area tradeoff. We similarly focus on the back-end execution engine in our study of clustered SMT temperature management. However, the important difference between our research and these studies on activity migration is that we do not require spare, backup resources for our DTM mechanisms. The extra register files, ALUs, and pipelines required by these prior schemes are inactive most of the time when there are no thermal emergencies, and they add significantly to the cost of the chip.

Skadron et al. [126] describe and use HotSpot (which thereafter became the standard tool for DTM research) to study a number of DTM techniques on a single-threaded core. Skadron [124] extends that work by combining DTM techniques in a hybrid scheme. Liao et al. [80][81] examine the impact of temperature dependent modeling of leakage power and thermal run-away on DTM techniques. That work advocates considering the temperature and voltage dependence of performance and power and the need for tightly coupled management of power and temperature. As in our work on clustered SMT processors, Powell et al. [99] use HotSpot to study the asymmetric usage of back-end processor resources and devise algorithms to spread the power and heating more evenly. However, they examine a single-threaded processor with one cluster, and thus only explore intra-thread diversity in application behavior.

Furthermore, we go beyond utilizing asymmetry to show how DVFS and our steering algorithms, which are well adapted to different workload characteristics, can be combined to provide a comprehensive DTM scheme. All these earlier efforts focus on single-threaded processors.

Other research looks at the more related topic of thermal management on SMT and CMP machines. Ghiasi and Grunwald [47] examine an asymmetric chip multiprocessor with a power hungry ILP intensive core and a spare low power, low performance core. Under normal operating conditions the high performance core executes a single thread. DTM consists of activity migration between these two cores which is done either proactively or reactively. Again, the need for a spare core results in a large amount of die area which is not used unless the workload is thermally intensive.

Donald and Martonosi [39] examine the thermal properties of two- and four-threaded SMT and CMP architectures. They find that these processors experience higher temperatures but have a similar heating distribution across their components to that of a single-threaded superscalar machine. Rather than using DTM, the authors propose to mitigate the high temperatures of the issue queue and result bus by changing the floor plan of the processor and by increasing the area of these units to spread out the heat. In another work, the same researchers investigate the possibility of exploiting varying application behavior to control temperature on an SMT processor [40]. Temperature guided fetch and rename policies are developed with the goal of restricting the flow of instructions from threads that are likely to heat the integer and floating point register files, which are deemed the hottest processor components. Like the work in Chapter 3, that paper tries to adjust the instruction flow in an SMT processor to mitigate the heating effects of a hot thread. However, both their policies throttle activity to reduce hotspots. Our static and dynamic steering techniques utilize

the differences in application thermal characteristics to eliminate hotspots, often without the need for throttling and reducing performance.

Heat Stroke is a denial-of-service attack on an SMT processor in which a malicious thread highly utilizes a particular pipeline resource, creating a hotspot on the chip [50]. Hasan et al. assert that such a Heat Stroke attack harms the performance of other threads on the processor when DTM mechanisms such as stop-go or DVFS slowdown the whole pipeline in order to cool the hotspot [50]. They conclude that it is imperative to isolate the offending thread and restrict only its execution through *sedation* which has the same effect as our Thread Dispatch Gating. We likewise propose thread and cluster specific DTM techniques to avoid penalizing threads that are not causing thermal problems. However, while the goal of their work is to stop malicious threads from degrading performance, our hot threads are behaving properly but happen to use the processor intensively. Rather than simply restrict the execution of these threads, we propose steering-based DTM policies which take advantage of the variations in applications to improve the thermal behavior of the hot thread by intelligently steering its instructions to clusters cooled by less resource intensive threads. This approach does not punish hot threads but instead increases IPC by lengthening the time before performance-harming DTM must be engaged.

Other research proposes HybDTM, a methodology for coordinating fine-grained hardware techniques and coarser grain software mechanisms to provide more effective thermal management [65]. The authors evaluate their approach on a Pentium 4 processor running Linux in single thread and SMT configurations. Like our work, their two level approach combines a low cost mechanism (an OS software technique) which tries to keep the processor as cool as possible and a higher cost mechanism (hardware stop-go) as a backup to prevent thermal emergencies. However, since they experiment with a real processor, they focus on making better use of features that are

already implemented whereas we propose new hardware DTM techniques for future clustered SMT architectures.

Heat-and-run exploits the variation in application resource usage to manage temperature in an architecture consisting of a chip multiprocessor composed of SMT cores [100]. By pairing complementary applications in a multiprogrammed workload, processor core resources are maximally utilized and maximally heated. These more efficiently used cores can then be cooled using activity migration across the cores of the CMP. While heat-and-run focuses on how to schedule threads to cores to maximize processor use under thermal constraints, our work examines how to manage a given set of threads within the core to reduce the performance cost of DTM.

Li et al. [74] compare the thermal behavior of a baseline superscalar processor to that of a two-way SMT and a CMP of two cores. They also compare the effectiveness of a variety of DTM techniques on these architectures, including DVFS and throttling of the fetch unit, rename, and the register file. Unlike our DTM policies, they do not employ mechanisms that adapt to application resource usage or exploit non-uniform thermal behavior to cool hotspots.

Donald and Martonosi [41] employ formal control theory to perform DTM in a four core CMP environment. They compare chip-wide and per-core implementations of a basic stop-go policy (equivalent to Global Dispatch Gating) and dynamic voltage and frequency scaling and experiment with migration of threads among the cores. In contrast, our work investigates in more detail the DTM opportunities within a single SMT core and shows that there are effective alternatives to DVFS at the per-core level. While we examine how non-uniform heating within an SMT core can be exploited for thermal management, they look at single-threaded cores and treat the cores holistically. Future research could explore how our intra-core techniques and

Donald and Martonosi's inter-core policies could work in tandem for even more effective DTM.

Chaparro et al. [30] also examine thermal management using stop-go, thread migration, and DVFS in a chip multiprocessor of single-threaded cores. However, these authors evaluate a sixteen-core architecture, compare a number of variants of stop-go and thread migration between cores, and focus on performing sensitivity studies to understand the impact of cool-down interval length, emergency threshold temperature, frequency of DTM decisions, and the quality of the thermal solution. Stavrou and Trancoso [130] look at DTM in future chip multiprocessors, identify undesirable chip heating scenarios, and then develop a thermal-aware scheduling policy that factors in spatial information such as the temperature of neighboring cores on the die and the cooling efficiency of different locations on the chip. Again, the insight these investigations develop for inter-core DTM policies can be combined with our techniques for managing temperature within the core.

Merkel and Bellosa [89] similarly look at preventing hotspots in a multiprocessor system. However, unlike most of the previous papers, they study real hardware and modify the Linux kernel load balancer, making it energy-aware, and schedule threads to even out power consumption of the cores. This work has similarities to our algorithms for DTM in a clustered SMT processor but their OS balancing algorithms operate between cores and at a much coarser granularity.

Chaparro et al. [28][29] propose that clustered microarchitectures naturally reduce temperatures in the processor because they distribute resources and computation among the back ends. They investigate single-threaded clustered processors and develop some simple steering mechanisms to mitigate thermal problems. The also propose cluster hopping, a form of activity migration where instructions are only sent to a subset of the back ends in the processor. Unlike our

simulated microarchitecture, their clusters are highly over-provisioned for a single-thread workload and thus unused clusters can again be thought of as spares. We study a whole new class of adaptive DTM techniques for clustered multithreaded machines which exploit workload non-uniformity and do not require spare or idle resources. Furthermore, we provide a comparison against DTM mechanisms such as DVFS and Global Dispatch Gating to demonstrate the benefits of our steering-based techniques. Finally, orthogonal research to ours explores the thermal benefits of front-end clustering [31]. The enhancements in that paper could be combined with our work to provide DTM for the entire processor.

2.3. Mitigating Hard Errors and Variations

Prior research on hard errors falls into several categories: (1) developing architectural models for manufacturing defects and lifetime wear-out and reducing the occurrence of these errors; (2) detecting the presence of permanent faults and isolating their impact; and (3) maintaining processor functionality despite the occurrence of an error. Prior papers on variations generally discuss techniques to model variability in microprocessors or mechanisms for reducing the impact of the variations. Research on each of these directions will be outlined in turn.

Srinivasan et al. [127] were among the first to look at lifetime reliability from an architectural perspective. They developed a model called RAMP for studying the impact of microarchitectural design decisions and runtime behavior on lifetime wear-out and proposed dynamic techniques to increase reliability. The same authors also looked at how microprocessor lifetime reliability will be impacted by technology scaling [128]. Kang et al. [63] develop a method for correlating changes in leakage power to increases in negative bias temperature instability (NBTI) degradation. Blome et al. [12] design an online hardware unit for the detection of gate oxide breakdown and use the unit to study this failure mechanism at the microarchitectural level. Feng et

al. [46] extend that work by using the wear-out detection units to guide the OS in a CMP to schedule jobs to intelligently manage lifetime wear-out. Paterna et al. [95] likewise try to slow the aging of multiprocessor system-on-a-chip architectures by dynamically controlling the duty-cycle of each core to balance their wear-out rates. Ramachandran et al. [104] compare two metrics for characterizing lifetime reliability, mean time to failure (MTTF) and a new metric, nTTF (the time to failure of n% of the population) and assess their value for microarchitects.

Austin [6] is among the first to develop a technique, called DIVA, for detecting hardware faults at the architectural level using simple checkers at the commit pipeline stage. Chatterjee et al. [32] continue to improve the checker to make it more performance efficient. Bower el al. [18][19] extend the capabilities of DIVA, adding mechanisms to isolate the faults, correct the errors, and deconfigure broken units. Distributed built-in self-testing and checkpointing techniques are devised by Shyam et al. [123] for detecting and recovering from defects. Meixner et al. [85][87] consider a different approach to error detection in simple cores which verifies that the four invariants of von Neumann-style processors hold during execution. Yilmaz et al. [141] focus on techniques to detect delay faults which cause timing errors in functional units. Schuchman and Vijaykumar [116] likewise focus on developing means for testing and isolating faults in the core logic. LaFrieda et al. [69] propose using dynamically coupled cores in a chip multiprocessor to provide fault detection through redundancy in a far more efficient manner than traditional static binding of core pairs. In the research of Chapters 4 through 7, we assume the use the above techniques for detecting and isolating faults in our chip multiprocessor, so that broken units can be deconfigured while maintaining processor functionality.

A number of papers develop schemes that tolerate permanent faults and allow the microprocessor to remain functional. Shivakumar et al. [122] are the first to

propose that the inherent redundancy in a processor can be exploited for hard error tolerance. Bower et al. [15][17] describe a new method of detecting and recovering from errors in processor array structures. Their mechanism uses spare rows in the structure which replace faulty ones that are mapped out. Srinivasan et al. [129] propose two methods to increase the processor lifetime: structural duplication and graceful performance degradation. Aggarwal et al. [2] study mechanisms for isolating faulty components in a CMP and reducing an error's impact through reconfiguration. Joseph [61] leverages hardware virtualization to salvage degraded cores by migrating computation or emulating instructions that cannot be supported due to failures. Finally, Meixner and Sorin [87] describe a similar technique for automatically modifying software in a way that maintains its functionality but changes the application's usage of the hardware to circumvent a faulty component. Many of the schemes for tolerating hard errors deconfigure broken components, keeping cores functional but operating in degraded states. The research presented in Chapters 4 through 7 examines the next stage of the problem: making most effective use of the resulting heterogeneous CMP through intelligent thread migration and global power management.

Significant prior research has looked at developing models for process variations in order to understand their impact at the architectural level. Romanescu et al. [111] make one of the first attempts to quantify how variability will affect microprocessor behavior. They then give an overview of their variations model in [110]. Humenay et al. [55][56] examine how parameter variations specifically impact multi-core chips. Das et al. [35][36] develop a process variations model to study how variability impacts yield and then develop a cache enhancement to shift more chips to high frequency bins. Bowman et al. [20] develop a statistical model for variations and an analytical throughput model for single-core and multi-core processors at the 22nm

technology node. They conclude that chip multiprocessors are inherently more tolerant to variability than a large monolithic architectural design. Lastly, Sarangi et al. [113] present their own model for process variations and a framework for estimating resulting timing errors.

Other studies have gone further and developed solutions for variability in different microarchitectural components. Agarwal et al. [1] examine how process variations cause failures in caches and propose a new cache architecture that is very effective at improving yield. Ozdemir et al. [94] also look at improving cache yield by developing microarchitectures that can disable cache ways and horizontal regions, as well as a variable latency design. In [77], Liang and Brooks develop a methodology for selecting microarchitectural parameters in a way that minimizes the impact of variations on frequency and instructions-per-cycle (IPC). Liang and Brooks also develop variable latency register files and execution units to prevent variability from impacting processor frequency [78].

Tiwari et al. [135] develop a technique to exploit the slack in faster pipeline stages and "donate" it to slower stages to compensate for variability in circuit timing. Teodorescu et al. [133] propose using dynamic fine-grain body biasing (D-FGBB) to adaptively change the voltage and frequency of a processor to meet yield specifications and maximize performance and power efficiency. Romanescu et al. [109] design architectural modifications that allow the processor to be clocked higher than some processor components can handle. They outline some microarchitectural changes to the register file, functional units, and first level caches that allow the processor to get around slower parts in these units. Liang et al. [79] show how variable latency units and voltage interpolation can be combined to provide a comprehensive defense against the effects of frequency variations.

2.4. Designed-Heterogeneous Chip Multiprocessors

In designed-heterogeneous CMPs [7][9][45][48][60][66][67][68][73[120], the heterogeneity is architected into the system rather than the unplanned result of hardware faults and variations. As a result, the degree and nature of heterogeneity is quite different than in the unpredictably heterogeneous CMPs that we study. Kumar et al. [66] show how significant power savings can be obtained by dynamically assigning single-threaded applications to cores with different power-performance tradeoffs in a heterogeneous CMP in response to changes in the application's behavior. In [68], the same authors focus on multiprogrammed performance and develop algorithms to schedule applications on cores that best match their execution requirements. However, since only two types of cores are used, the solution space is small and thus a simple sampling scheme achieves good assignments. Becchi and Crowley [9] extend that work to use performance driven heuristics for scheduling. Our reliability scheduling problem is far more complex: a huge number of unpredictably heterogeneous organizations can arise in terms of frequency, dynamic power, and leakage currents, in addition to architectural parameters. In a third paper, Kumar et al. [67] study heterogeneous architectures where the cores are not restricted to a few configurations. Their goal is to determine how much heterogeneity is necessary and how the cores should be designed to fit a given power budget. Here, they focus on the design issues rather than the scheduling aspect that is the focus of our research.

Balakrishnan et al. [7] study the impact of asymmetry in core frequency on parallel commercial workloads using a hardware prototype. Li et al. [73] also study asymmetric multi-core architectures on real hardware, in this case SMP and NUMA systems with frequency heterogeneity emulated using clock duty cycles. Ghiasi et al. [48] examine heterogeneity due to cores running at different voltages and frequencies. While their work only adapts the cores' voltages and frequencies, we investigate both

thread scheduling and global power management and the scalability of these algorithms. Furthermore, our CMPs are significantly more heterogeneous because we model leakage variation and hard errors in addition to frequency variability.

Fedorova et al. [45] develop a reinforcement learning algorithm that schedules threads on heterogeneous systems to simultaneously optimize performance, enforce fair CPU sharing, and load balance among the cores. Shelepov et al. [120] design a static heterogeneous scheduling algorithm, which employs architectural signatures imbedded in the application binary that encode the memory-boundedness of the program, allowing the applications to be effectively matched to frequency asymmetric cores without online profiling. Likewise, Jooya et al. [60] study static and dynamic scheduling techniques, and show that a dynamic approach that finds favorable pairings based on tracking the history of the applications' behaviors on the cores, outperforms static techniques.

2.5. Early Work on Unpredictably Heterogeneous Chip Multiprocessors

Recent work has begun to address the challenge of reducing performance and power efficiency losses in the face of hard errors and variability in chip multiprocessors. Sylvester et al. [131] are the first to argue for developing a holistic approach for managing unpredictable silicon where circuit-, microarchitecture-, and system-level techniques coordinate to tune performance, monitor reliability, and manage self-healing mechanisms. Roberts et al. [108] explain how unpredictable heterogeneity can develop in a designed-homogeneous multi-core processor, and propose and analyze the use of linear programming to maximize performance on a three-core processor with different operating frequencies. Independently and in parallel to our research described in Chapter 4, Bower et al. [16] highlight the need for intelligent runtime scheduling policies for unpredictably heterogeneous processors and discuss key issues that must be factored into any solutions. The above studies provide

a strong rationale for the need to develop adaptive thread management techniques specifically for unpredictably heterogeneous CMPs but do not delve deeply into possible solutions. Donald and Martonosi [42] develop an analytical model for studying the power-performance efficiency of chip multiprocessors impacted by variations and apply Amdahl's Law to parallel applications running on these processors.

Teodorescu and Torrellas [134] is the only work to our knowledge to consider both scheduling *and* power management via dynamic voltage and frequency scaling (DVFS) in a multi-core microprocessor with different frequencies due to process variations. They conduct a design exploration, propose a number of schedulers to satisfy different objectives, and develop a linear programming solution which improves on the MaxBIPS algorithm of [58]. The scheduling algorithms we propose in Chapter 4 differ from theirs because we consider both process variations and hard errors and optimize for power and performance simultaneously. In Chapter 7, we employ their VarF&AppIPC scheduler and their linear programming method, LinOpt, as baselines for our study of scalable runtime management algorithms. Finally, Herbert and Marculescu [53] also tackle power management (but not scheduling) under process variations. They develop a greedy DVFS algorithm that addresses both the frequency and the leakage costs of variations. In order to be comprehensive, we develop a representative greedy power management algorithm in the scalability study of Chapter 7.

In summary, our work is distinct from prior work on unpredictably heterogeneous CMPs in that (1) we are the first to compare the scalability, performance, and power efficiency of a wide range of potential scheduling and power management algorithms for many-core systems with hundreds of cores; (2) we consider degradations due to both hard errors and variations, which adds an extra

dimension of complexity to the scheduling and power management problems; (3) we experimentally prove for the first time that coordination between the power manager and scheduler is unnecessary for many-core processors and explain the reasons for this result; and (4) we propose and evaluate new highly scalable scheduling and power management algorithms for many-core systems with degradations due to both hard errors and variability.

CHAPTER 3

ADDRESSING THERMAL NON-UNIFORMITY IN SMT WORKLOADS

3.1. Introduction

Dynamic thermal management (DTM) has been an active area of microarchitecture research for the past several years. While many DTM techniques have been proposed and studied for single-threaded, SMT, and CMP machines, dynamic voltage and frequency scaling (DVFS) stands out as the most effective approach [30][41][74]. DVFS reduces both the voltage and frequency of the processor, enabling an almost cubic reduction in dynamic power as well as super-linearly decreasing static power, making it difficult to beat. While DVFS proves to be most effective when cooling a uniformly hot, CPU-intensive workload, it is much less effective when there are differences in the thermal behavior of the simultaneously running threads. Specifically, in an SMT processor, DVFS can penalize the performance of one thread in order to cool a hotspot caused by another. In workloads where one thread is very CPU-intensive and the other is not, or in mixed floating point/integer application workloads, engaging DVFS on an SMT core can often force the other non-offending thread to slow down despite the fact that it is not responsible for the hotspot.

To illustrate this phenomenon, we examine an SMT workload consisting of two SPEC CPU2000 benchmarks: *vortex*, a CPU intensive integer application, and *equake*, a less intensive floating-point application. Figure 3.1 shows the baseline performance of the pair on the clustered SMT architecture we model (described in detail later) when no dynamic thermal management is used. In comparison, Figure 3.2 shows the performance when DVFS is used to cool the processor. (In Figure 3.1, Figure 3.2, and Figure 3.12, performance is measured as billions of instructions per second (BIPS), averaged over 100K intervals.) The periodic performance degradation

in Figure 3.2 is due to intervals during which DVFS is employed. Figure 3.3 illustrates the temperatures of the hottest components of the processor during the DVFS simulation. Because of *vortex*'s intensive behavior, the hotspots are consistently the integer ALU and integer multiplier unit on its clusters. Dips in temperature in Figure 3.3 from engaging DVFS correspond to the performance degradation points in Figure 3.2. Clearly, *equake* is being penalized by DVFS even though its portion of the processor is nowhere near the thermal danger limit of 87ºC. This example illustrates how *thermal non-uniformity* among the simultaneously running threads can increase the performance degradation of DVFS-based DTM by penalizing a cool thread that runs on the same core as the hot one. A better DTM policy would intelligently manage individual threads and take advantage of the non-uniformity in temperature to eliminate chip hotspots.

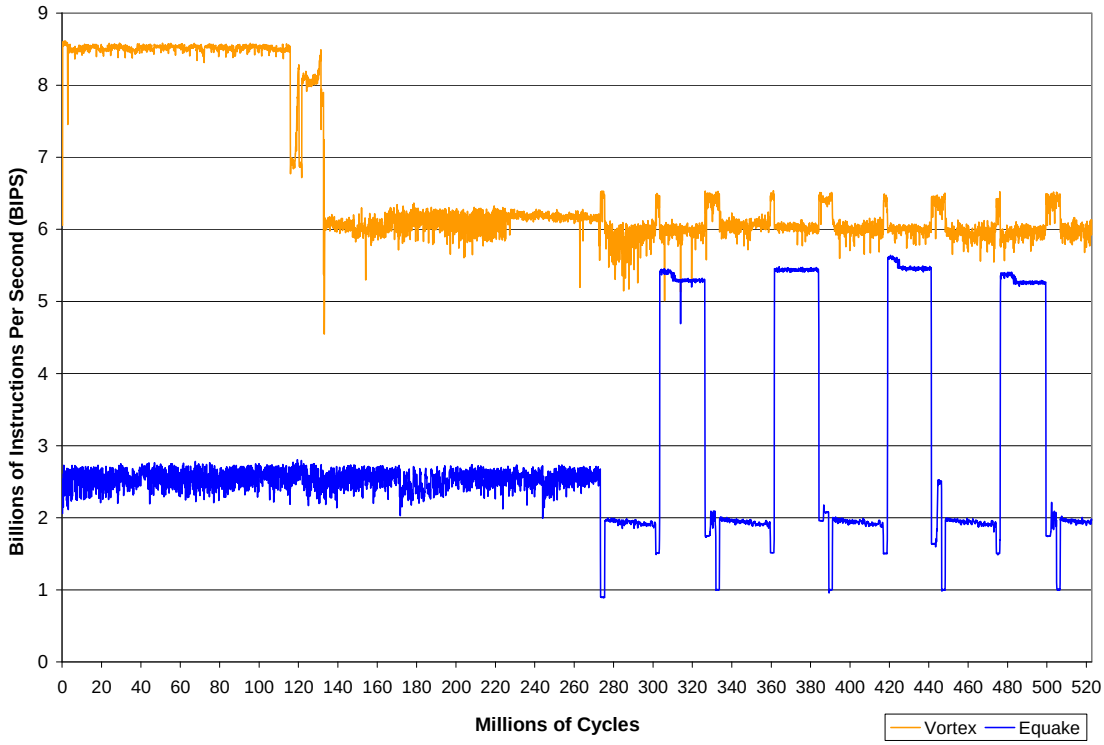


Figure 3.1: vortex / equake – performance with no DTM mechanism.

In this chapter, we develop effective alternatives to DVFS for thermally non-uniform SMT workloads. We propose DTM techniques that utilize the inherent steering mechanism in a clustered SMT microprocessor [34][44][70] to exploit thermal non-uniformity among the threads to cool the chip more efficiently. For workloads composed of threads with non-uniform heating characteristics, our best DTM algorithm, Counter-Based Steering, prevents all thermal violations with a worst case performance of 1% compared to 6.4% for DVFS.

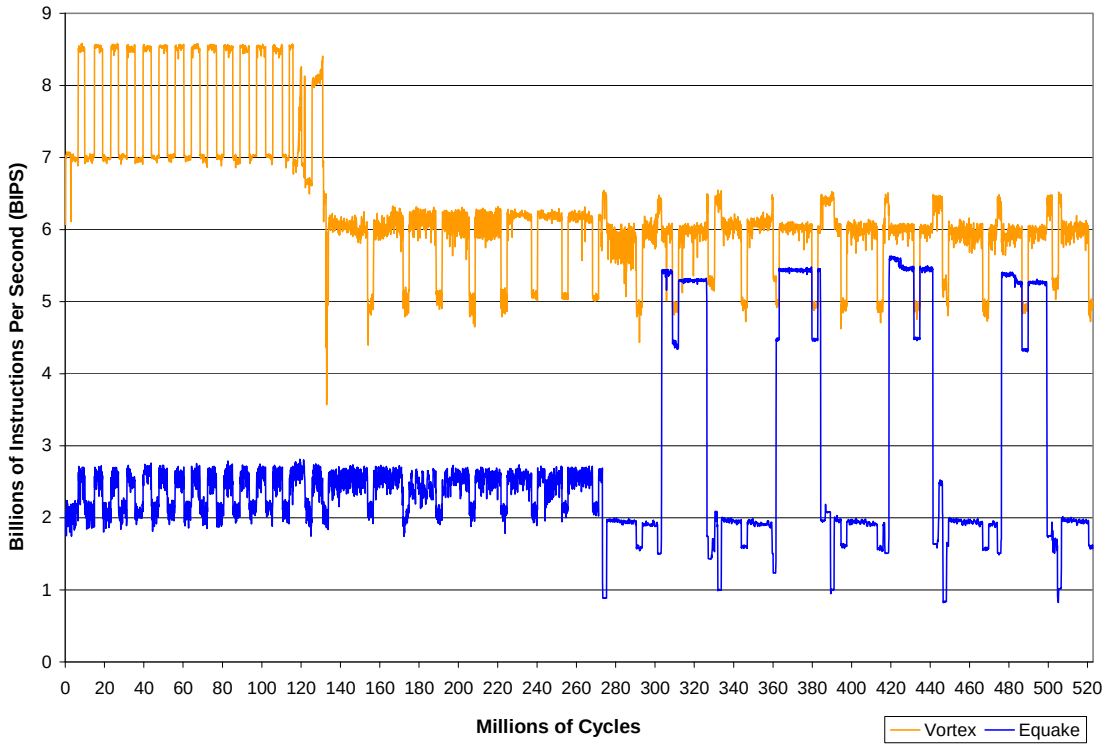


Figure 3.2: vortex / equake – performance with dynamic voltage and frequency scaling.

We also propose a "best of both worlds" DTM policy which utilizes the complementary properties of both steering- and DVFS-based algorithms. Namely, the steering-based mechanisms exploit the thermal differences of the running threads to reduce hotspots and avoid employing DVFS (and slowing down all threads) when possible, whereas DVFS is effective in cases where a set of "hot threads" are

uniformly heating the back ends (leaving little opportunity for the steering-based mechanisms to exploit temperature differences among the clusters). Moreover, adding steering-based DTM to DVFS requires only minor changes to the baseline clustered SMT organization.

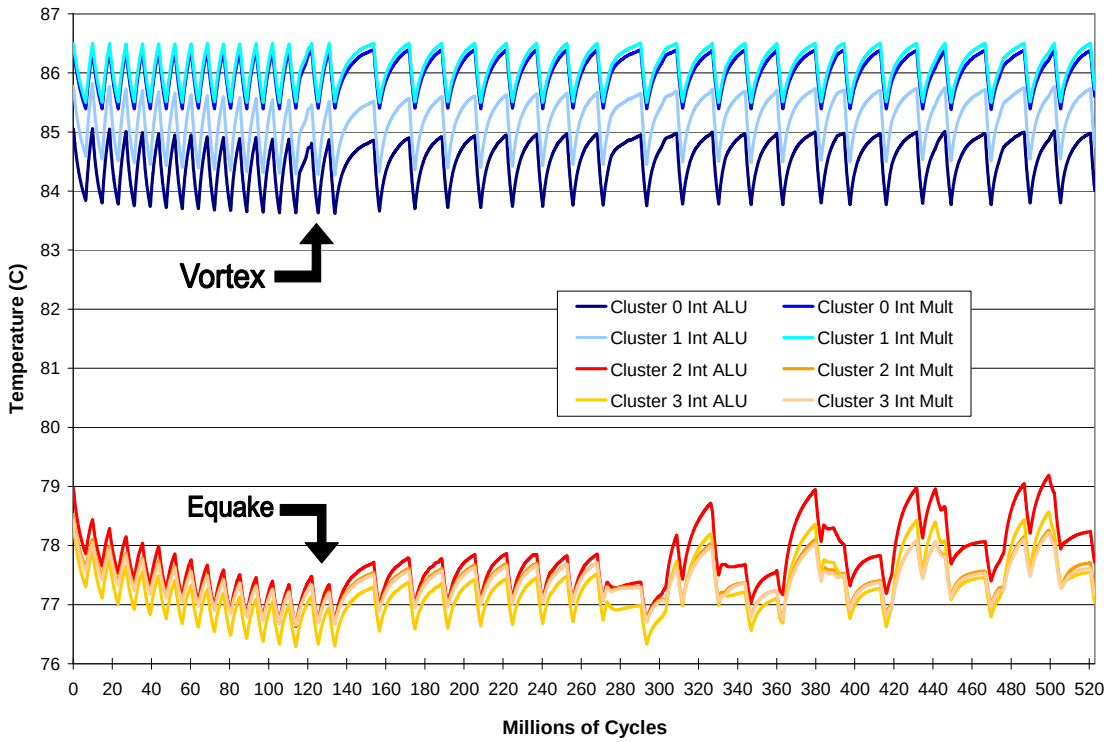


Figure 3.3: vortex / equake – temperature with dynamic voltage and frequency scaling.

Our research makes a number of novel contributions to the field of dynamic thermal management. This is the first work to explore DTM policies for clustered SMT microarchitectures, a unique design that combines the benefits of both multithreaded and multi-core architectures. We discuss an overlooked drawback of dynamic voltage and frequency scaling, which is that engaging DVFS on an SMT core will unfairly penalize threads that are not causing the thermal emergency. We show that this is a particularly large problem for non-uniform workloads and propose a novel thermal management technique to address this deficiency. Our algorithm

exploits the spatial non-uniformity in temperature *within* a processor core caused by differences in thread pipeline usage to provide very low cost DTM. Finally, we devise a new kind of thermal management policy that combines two distinct techniques – DVFS and Counter-Based Steering, each specialized for a particular class of benchmarks – to provide effective DTM across the full range of multithreaded workloads.

3.2. The Clustered SMT Microarchitecture

The clustered SMT processor, shown in Figure 3.4, is similar to that explored in [44]. The execution core, consisting of the issue queues, register files, and the functional units, is divided into multiple clusters with communication paths between the unified front end and the back ends, the back ends and the L1 data cache, and among the back ends for passing operand values. A traditional SMT front end is used to fetch, decode, and rename instructions from multiple threads, and a *steering mechanism* is employed to assign these instructions to back ends. While sharing of back ends among multiple threads is possible, prior research [44][70][102] has shown that the best performance and power characteristics are obtained by largely isolating the threads from each other by assigning them to separate cluster groups. In order to reduce the performance cost of inter-cluster communication of operand values, instructions from the same thread are usually assigned to a contiguous group of adjacent clusters.

In our architecture, we found that the back-end clusters were the hottest part of the die and focused on alleviating hotspots in this section of the processor. This result is supported by previous research showing that register files and execution units are typically the biggest thermal concern [39][40][41][51][74][126]. However, in other designs the front end could also be a source of thermal emergencies. In [31], the authors explore partitioning the front end to control temperature and the architectural

enhancements they propose can be combined with our techniques to provide thermal control across the processor.

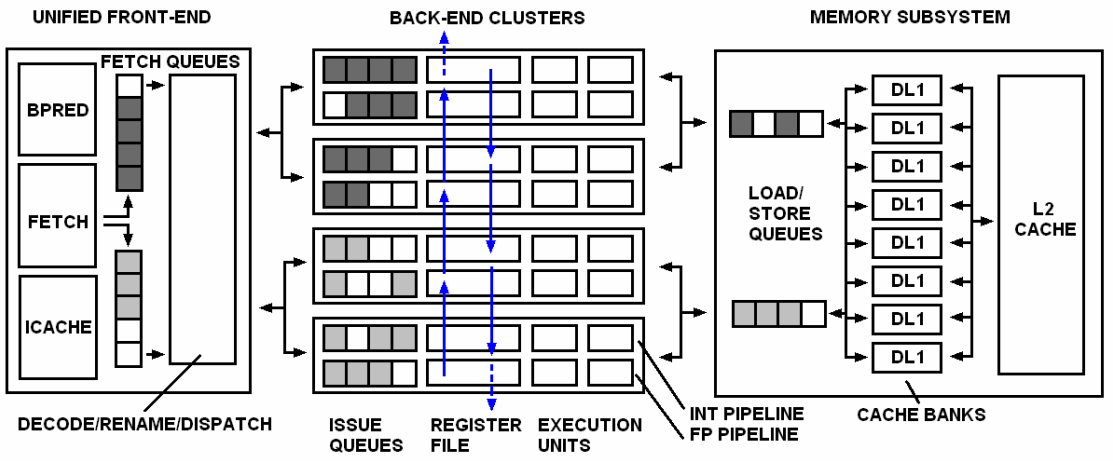


Figure 3.4: A clustered simultaneous multithreaded microarchitecture.

Clustered SMT processor designs may appear to go against the current trend towards chip multiprocessors containing many simple, perhaps single-threaded, cores. However, sequential code will not disappear altogether. Some workloads will contain high levels of instruction-level parallelism that is most effectively processed by wide-issue cores, some applications may be extremely difficult to parallelize, and there will always be some programs that software developers have not yet parallelized. Clustered SMT processors provide the flexibility to address these sequential workloads – by providing a low complexity, wide-issue engine when needed – as well as supporting parallel workloads through multithreading. The partitioning of back-end resources greatly reduces the complexity of the design, as well as the power and temperature. In addition, there is industry precedence for designing wide cores with back-end clusters such as the Alpha 21264/21364, IBM Power4, and the SMT IBM Power5. Clustered SMT cores could be implemented in CMPs, along with narrower cores in an asymmetric configuration, providing a design that is performance and power-efficient across a large variety of applications.

Moreover, a clustered SMT microarchitecture provides a natural platform for addressing non-uniformity for several reasons. First, the clustering of the hottest components on the die, namely the functional units, register files, and issue queues, into multiple back-end clusters, permits threads, and even different instructions from the same thread, to be largely thermally isolated from other threads or instruction groups, yet run simultaneously. This is in contrast to a traditional SMT microarchitecture in which threads largely share these hot back-end resources. Second, a clustered SMT microarchitecture has a built-in communication mechanism to permit instruction operands to propagate to the cluster in which they are needed. This allows a thread's register values to rapidly move from one cluster to another with low overhead and no additional support. Finally, the steering mechanism in a clustered SMT provides a simple, yet effective means for temperature management when a thermal emergency arises. If a particular thread, or a subset of that thread's instructions, causes a thermal emergency in a particular back end, then those instructions can simply be steered to a cooler back end for some period of time. Other instructions, possibly from a different thread with less stringent cooling requirements at this moment, may begin to be steered to the hot cluster to make use of its resources while instructions drain from the prior thread.

In this chapter, DTM mechanisms are evaluated for a two thread, four cluster microarchitecture. Each back-end cluster is dedicated to a single thread at any time, except for the temporary overlap that occurs when cluster assignments are switched by the steering algorithm. Nominally, each thread is allocated two back ends in the processor. This simplifies the microarchitecture implementation and allows our DTM policies to take advantage of the differences in the thermal heating profiles of the applications. Furthermore, we found that back-end resource utilization was very high,

indicating that having an additional thread competing for issue queue slots, issue bandwidth, etc. would not be beneficial.

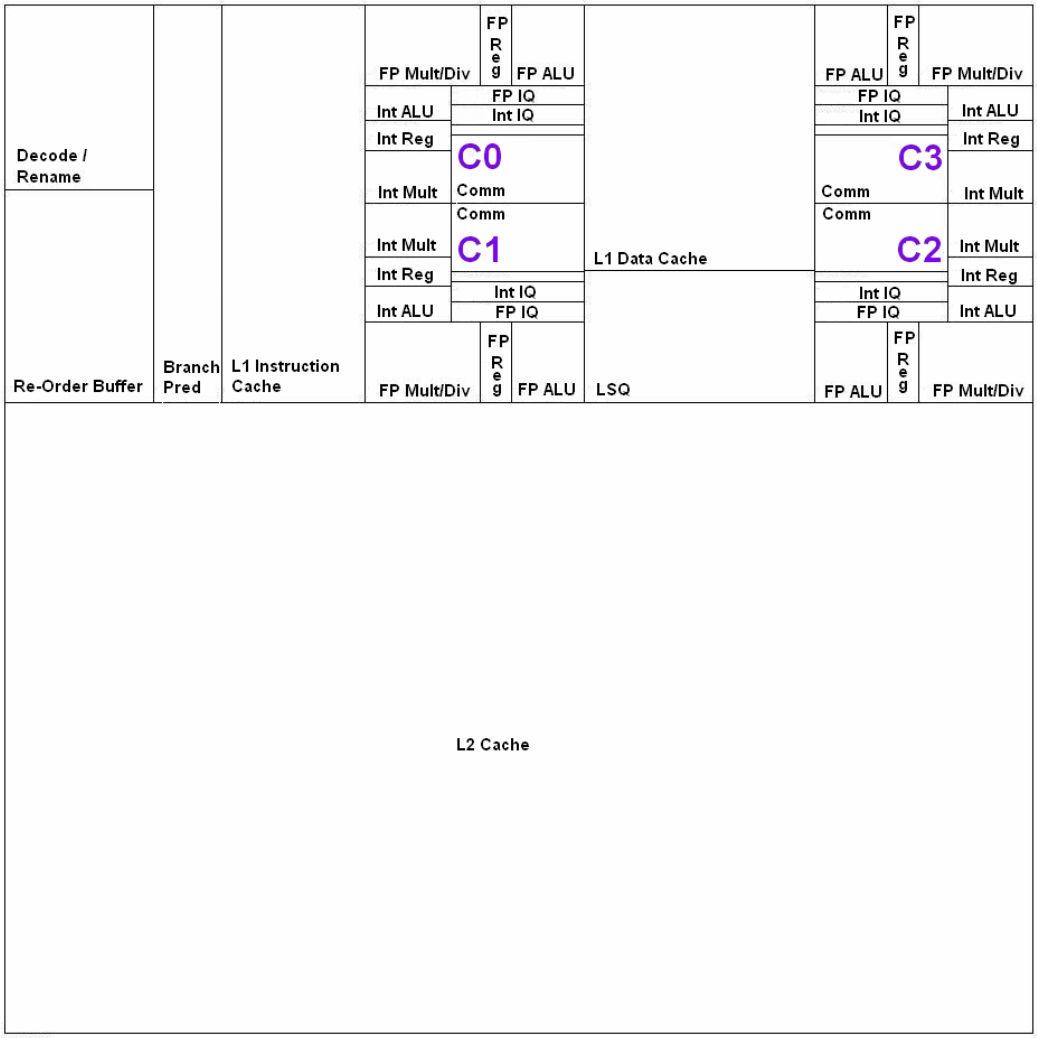


Figure 3.5: The floor plan of the clustered SMT microarchitecture.

Figure 3.5 illustrates the floor plan of our clustered SMT design. The back-end clusters consist of integer and floating point issue queues, register files, and execution units, as well as a shared set of communication links and a register access window for inter-cluster operand passing [143]. The communication links provide a ring interconnect for forwarding register values to other back ends, which can forward two values in either direction along the ring each cycle. In addition, there are point-to-point

links between each back end and the front end for dispatching instructions, accessing the data L1 cache for stores and loads, communicating branch results, and updating the re-order buffer. Again, the links can send two results in each direction every cycle. The contention and power of all the communication links are modeled. We assumed a centralized, banked L1 data cache in order to simplify the architecture and to allow us to analyze steering policies without the affect of relocating threads away from their associated cache lines. We placed the data cache and the load store queue in the middle of the clusters on the die to permit a single cycle hop to the cache for loads and stores.

Table 3.1: Simulated clustered SMT microarchitectural parameters.

Unified Front-End Parameters	
Fetch bandwidth	8 instructions per thread
Branch predictor type	Hybrid of bimodal and 2-level
Bimodal predictor entries	2048, 2-bit counters
Level 1 predictor table entries	1024, history of 10 branches
Level 2 predictor table entries	4096, 2-bit counters
BTB entries	2048, 2-way associative
Branch misprediction penalty	11 cycles
Fetch queue size	16 entries per thread
Decode/rename/dispatch/commit bandwidth	8/8/8/8 instructions per thread
Re-order buffer size	200 entries per thread
Back-End Cluster Parameters	
Integer/FP issue queue size	20 entries each
Integer/FP register file size	80 entries each
Integer/FP simple ALU	1 of each
Integer/FP complex ALU w/ multiply & divide	1 of each
Register access window	10 slots
Inter-cluster communication links	2 point-to-point bi-directional links around the ring
Front-end communication links	2 point-to-point bi-directional links to the front end
Memory Hierarchy	
Load/store queue size	64 entries per thread
L1 instruction cache	64KB, 4-way, 32B blocks, 8 banks, 1 cycle latency
L1 data cache	64KB, 4-way, 64B blocks, 8 banks, 2 cycle latency
L2 unified cache	4MB, 8-way, 128B blocks, 16 cycle latency
Memory latency	100 cycles

The front end uses the ICOUNT fetch policy [136] to determine the number of instructions to fetch for each thread each cycle. The baseline front to back end performance-based steering mechanism statically dedicates two clusters to each thread [44]. As in [8], steering of instructions within the same thread to its two clusters is based on load balancing and the location of the instructions' operands. The criticality of each source operand is also considered and ties are broken by sending the instruction to the cluster whose operand is predicted to be produced last [8]. Further details of the microarchitecture can be found in Table 3.1.

3.3. Methodology

In this chapter, the simulation infrastructure is based on the SimpleScalar 3.0 architecture simulator modeling the Alpha ISA [24], augmented with Wattch [23], Hotspot 2.0 [126], and HotLeakage [142] for modeling dynamic power, temperature, and static power, respectively. The simulator has been further modified to support the clustered SMT microarchitecture and dynamic thermal management.

We assumed a processor implemented in 70 nm technology with a clock frequency of 2.5GHz and a 1.0V supply voltage. We use HotSpot's default parameters for the thickness of thermal package components including the die-to-spreader thermal interface material (0.075 mm), the heat spreader (1 mm), and the heat sink thickness (6.9 mm). The ambient temperature is set to 45°C [126]. The *thermal hard limit*, which is the temperature the microprocessor must not exceed to operate properly, is set at 87°C to be consistent with ITRS projections for maximum junction temperature for the 70 nm technology node [117].

The convection resistance of the heat sink models the quality of the thermal package. It should be set so that with appropriate thermal management, the processor's performance is not degraded severely under worst-case conditions, and so that DTM is not required in the average case. We used a convection resistance of 0.40 K/W for our

clustered SMT microarchitecture. One problem with HotSpot is that it inaccurately models the heat flow through the edges of the die [125]. As a result, the blocks near the edge of the chip, particularly the units of cluster three in the corner of the chip (see Figure 3.5), were disproportionately hot relative to their power dissipation. As suggested by [125], we addressed this problem by surrounding the die with a ring of dummy blocks about 1mm x 1mm which allowed floor plan units on the edge to cool more reasonably.

Our leakage power model is based on the code provided for download with HotLeakage [142]. However, we have extended HotLeakage significantly, from the original modeling of the caches and register file, to include static power for all front- and back-end components of the processor. This extension is based on the leakage power estimation method presented by Butts and Sohi [26] which employs the following main equation:

$$P_{static} = V_{dd} \cdot N \cdot k_{design} \cdot \hat{I}_{leak}$$

Here, P_{static} is the static power of a block in the HotSpot floor plan. V_{dd} is the current processor supply voltage. N is the number of transistors in that processor component, calculated using the area of the floor plan block multiplied by transistor densities for logic and SRAM structures given by [117]. The multiplicative factor k_{design} accounts for circuit design parameters such as transistor stacking, sizing ratios between the NMOS and the PMOS transistors, and the type of circuit being used. We used a combination of the k_{design} values presented in a chart in [26] for each processor block, appropriately matching the types of circuits found in that unit. For example, the issue queues are assumed to be a combination of CAM cells for the wakeup component, SRAM cells to store instruction information, logic for the instruction select stage, and multiplexers for the write-back component and thus the k_{design} value

used is the weighted average of these. On the other hand, the execution units are assumed to be pure logic and thus have a k_{design} of 11. Finally, $\hat{I}_{leak}$ is the average leakage current of a single transistor for a given technology and temperature.

We model temperature dependent static power by taking the temperature that HotSpot generates every 10,000 cycle interval and recalculating the $\hat{I}_{leak}$ current term for each block for the next interval. Since we use such a short interval, there is no need to perform iterative leakage calculations because temperatures can only change a few hundredths of a degree in that time frame. HotSpot calculates the starting temperature of the next interval, based on the current interval's temperature and the sum of the dynamic and static power generated over this interval.

When conducting DTM architecture simulations, it is critical to obtain realistic starting temperatures for the components of the thermal model and in particular the heat sink temperature [126]. This is achieved by fast-forwarding four billion instructions per thread, running each benchmark for 500 million instructions on our baseline architecture without DTM, and then using HotSpot to calculate steady-state temperatures for each block. As in [126], these steady-state temperatures are scaled so that no block exceeds the emergency threshold and then used as a starting temperature for the DTM simulations. Each DTM simulation also consists of fast-forwarding four billion instructions per thread, and then the simulation is allowed to run for 400 million instructions in order to warm-up the caches and branch predictors and allow the DTM mechanisms to begin operating. With the simulation now in a realistic performance and temperature state, each benchmark is executed for 500 million instructions during which time we collect our results. Once a thread completes its 500 million instructions, it continues to run without recording statistics in order to maintain realistic temperature conditions for the other thread and permit continued use of the DTM techniques. Overall performance is measured using the harmonic mean of

the IPCs for DTM simulations compared to a baseline run consisting of the same steps but which uses no DTM mechanism. The harmonic mean was chosen to prevent rewarding algorithms that unfairly constricted the performance of one application in favor of another.

We compare our steering-based DTM policies to dynamic voltage and frequency scaling to illustrate their effectiveness against the most popular contemporary approach. DVFS scales the overall processor supply voltage and frequency until the temperature cools to an acceptable level. By reducing voltage and frequency, the dynamic power of the processor is reduced almost cubically due to dynamic power's linear dependence on frequency and quadratic dependence on voltage. Static power is also reduced significantly because as described in the equation above, P_{static} is linearly proportional to V_{dd} and the $\hat{I}_{leak}$ term also has a supply voltage dependence [142].

Our implementation of DVFS only employs two voltage levels, the nominal and the low voltage level, as advocated by Skadron [124], who showed that for DTM, there is virtually no benefit to using multiple voltage steps. After exploring a wide range of low voltages, we found that 0.8V had the best performance and successfully prevented all thermal violations. At this low voltage the processor frequency was calculated to be 2.055 GHz by using the following equation [103]:

$$frequency = k \cdot (V_{dd} - V_t)^{\alpha} / V_{dd}$$

Here V_{dd} is the supply voltage, V_t is the threshold voltage (0.18V) [117], α is a technology dependent constant set to 1.5, and k is a fitting constant set to 3.366 in our simulations so as to match our nominal voltage (1.0V) and frequency (2.5 GHz).

The benchmark sets for our simulations were selected using the 18 hottest SPEC CPU2000 integer and floating point applications. Each benchmark is

represented as equally as possible in the mixes and care was taken to evenly combine benchmarks so that sets had different mixes of floating point and integer applications, high and low IPC applications, and hotter and colder applications to provide varied workloads for our simulations. One exception is that two low IPC, low heat, benchmarks are not combined, as that pair would lead to a simulation without any need for DTM.

Table 3.2 outlines our benchmark sets and their type, performance, and thermal characteristics. It also shows the average number of clusters in thermal violation during the execution of the applications when no dynamic thermal management is employed. For example, a value of 50% means that on average half the clusters are in thermal violation during the run. In addition, the peak temperature reached during the run without DTM indicates how far the application pairs would exceed the emergency thermal threshold of 87°C.

Table 3.2: Baseline benchmark set performance and thermal characteristics.

Benchmark Pairs	**Average Number of Clusters in Violation**	**Peak Temperature**	**Type – IPC (INT/FP – High/Low)**	**Temp (Hot/Warm)**
Uniform Workloads				
applu / apsi	87.84%	94.41ºC	FF-HH	HH
bzip2 / vortex	100.00%	92.74ºC	II-LH	WH
eon / galgel	100.00%	92.70ºC	IF-HH	HH
facerec / mesa	100.00%	92.68ºC	FF-HH	HH
gzip / vpr	50.00%	89.84ºC	II-HL	WW
Non-Uniform Workloads				
ammp / lucas	24.65%	87.84ºC	FF-HL	HW
gcc / mgrid	100.00%	93.67ºC	IF-LH	WH
mesa / parser	50.00%	94.91ºC	FI-HL	HW
swim / wupwise	88.22%	90.55ºC	FF-LH	WH
vortex / equake	50.00%	93.86ºC	IF-HL	HW

In spite of the diversity among individual SPEC benchmarks, the most important factor influencing the effectiveness of the DTM techniques on a benchmark pair is the uniformity of the threads' thermal and performance characteristics. The top five pairs in Table 3.2 consist of uniform workloads where both benchmarks are high IPC *and* high temperature, or both benchmarks are integer applications which thermally stress the same components of the processor. We will show in Section 3.5 that DVFS is the most proficient DTM technique for dealing with this kind of application mix. The bottom five benchmarks are mixed floating point and integer application pairs or two floating point applications, where in either case, one benchmark runs significantly cooler and uses resources less intensively than the other. We focus on these non-uniform benchmarks in the next section, and demonstrate that our fine-grain, clustered SMT temperature management policies are far more effective than DVFS on this workload type.

3.4. Steering-Based Dynamic Thermal Management Policies

This section explores a number of the steering-based DTM policies. All the mechanisms adhere to the same general framework. When not actively addressing a thermal emergency, the architecture employs the baseline performance steering policy described previously. Simultaneously, the DTM techniques monitor temperatures within each cluster every 10,000 cycles. If the temperature of a back-end component reaches the *trigger threshold*, then the dynamic thermal management mechanism reacts in some manner. When the temperature of the component drops to the *stop threshold*, the thermal emergency is considered alleviated. At this point, if the DTM technique involves throttling some active component, such as avoiding the sending of instructions to a hotspot, the technique is disengaged and processor operation returns to normal. Note that all of the proposed temperature control mechanisms successfully prevented the occurrence of thermal violations in the back-end clusters.

Our DTM policies can be divided into two general types. Section 3.4.1 describes *dispatch gating policies* which reduce heat by decreasing processor activity. These mechanisms guarantee that a particular maximum temperature will not be exceeded but incur a high performance penalty. Section 3.4.2 presents *heat spreading policies* which seek to balance heat dissipation among the clusters to reduce the occurrences of hotspots. These mechanisms incur a negligible performance cost but cannot guarantee a safe temperature under all circumstances. Thus, they require a backup fail-safe mechanism such as dispatch gating. When the spreading policies are successful, however, dispatch gating is not needed, thus avoiding the performance penalty.

3.4.1. Dispatch Gating Policies

The dispatch gating policies all build on the following basic mechanism. The *dispatch gating trigger threshold* is set to 86.5°C, 0.5°C below the thermal limit, to give the thermal management mechanism some breathing room to operate and to budget for possible temperature sensor error. The value of 0.5°C was empirically determined as the closest value to the emergency threshold that still guaranteed that dispatch gating would succeed in keeping the temperature at a safe level. Moreover, 0.5°C should be a sufficient margin to account for temperature sensor error considering recent advancements in on-chip CMOS sensors [33][59][96]. With dispatch gating engaged, no further instructions are sent from the front end to the issue queues of the hot back end. The hot cluster, receiving no further instructions, will soon run out of work and begin to cool down. In addition, once all the instructions have passed through the cluster's pipeline, dispatch gating clock-gates that back end's resources. This stops all switching activity and thus eliminates the dynamic power of the cluster. Furthermore, to eliminate the leakage power, all structures are power-gated except the register file, which must be left on to preserve register values possibly

needed in the future. When dispatch gating succeeds in lowering the peak temperature to the *stop threshold* (85.5°C), the thermal emergency is considered averted and gating is disengaged. Lower values for the stop threshold were considered, such as 84.5°C and 83.5°C, in the hope that they would decrease the probability that the hotspot would quickly reheat and require another DTM response causing a ping-pong effect. However, we found as in [30] that hotspots cool with an exponential curve such that keeping dispatch gating on for longer intervals cools less efficiently and reduces performance because the DTM mechanism remains on for more of the execution time despite being engaged less often.

The difference between the dispatch gating policies is the granularity at which the back end can be gated. The simplest policy is Global Dispatch Gating, and consists of ceasing dispatch to all the clusters, making it representative of pipeline throttling or resource toggling [22][74][126]. This policy emulates a thermal management scheme in a non-clustered architecture. Thread Dispatch Gating distinguishes between the thermal activities of different threads. This technique stops dispatching instructions from a hot thread to the execution engines but permits other threads to proceed normally.

Cluster Dispatch Gating utilizes the clustered nature of our microarchitecture by halting the dispatch of instructions to a specific hot cluster. The thread whose cluster is disabled steers all instructions to its remaining cluster until the thermal emergency subsides. The ability to control the flow of instructions to different parts of the back end is a feature unique to architectures with clustered back ends and enables finer grained DTM control. Figure 3.6 illustrates how the three dispatch gating policies operate in the presence of a hotspot in one of the clusters of thread 1.

We compare our steering-based techniques to dynamic voltage and frequency scaling, another heat reduction mechanism that does not require gating. DVFS is

employed similarly to Global Dispatch Gating, turning on when the hottest part across all back ends reaches the trigger threshold of 86.5°C, and turning off when the stop threshold of 85.5°C, is reached. We assumed that the processor pipeline must stall for 10µs whenever the voltage and frequency is changed, which is consistent with previous research [22][74][124][126].

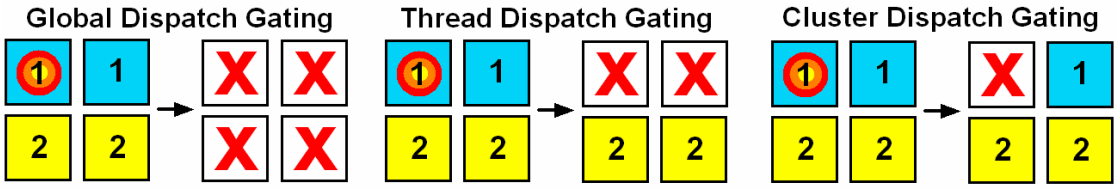


Figure 3.6: The three dispatch gating techniques.

3.4.1.1. Results

Our temperature results show that all dispatch gating policies were successful at preventing thermal emergencies. Therefore, we focus on the difference in performance degradation incurred by the applications as a result of running each policy. Figure 3.7 shows the performance of DVFS and the three dispatch gating techniques relative to the baseline architecture with no dynamic thermal management for the five non-uniform workloads. On average, Global Dispatch Gating causes a 12.7% performance degradation, compared to Thread Dispatch Gating with a 7.5% penalty, and Cluster Dispatch Gating with a 5.0% penalty. Clearly, it is beneficial to dispatch gate at a finer granularity. Due to differences in processor resource utilization and heating between a workload's benchmarks, it is beneficial to employ Thread Dispatch Gating which is able to isolate and cool the thread with the hotspot while allowing the other thread to execute normally when it is not also overheating. The dependence-based baseline performance steering algorithm tries to dispatch dependent instructions to the same cluster as the instructions producing their operands. As a result, back-end clusters of the same thread may exhibit quite different temperature

characteristics. Cluster Dispatch Gating benefits from cutting power to only the hot cluster, allowing the thread to continue to make forward progress using its other cluster as long as that back end stays cool. While Cluster Dispatch Gating has lower performance than DVFS on three out of the five workloads, DVFS is still better overall with a slowdown of 4.1%.

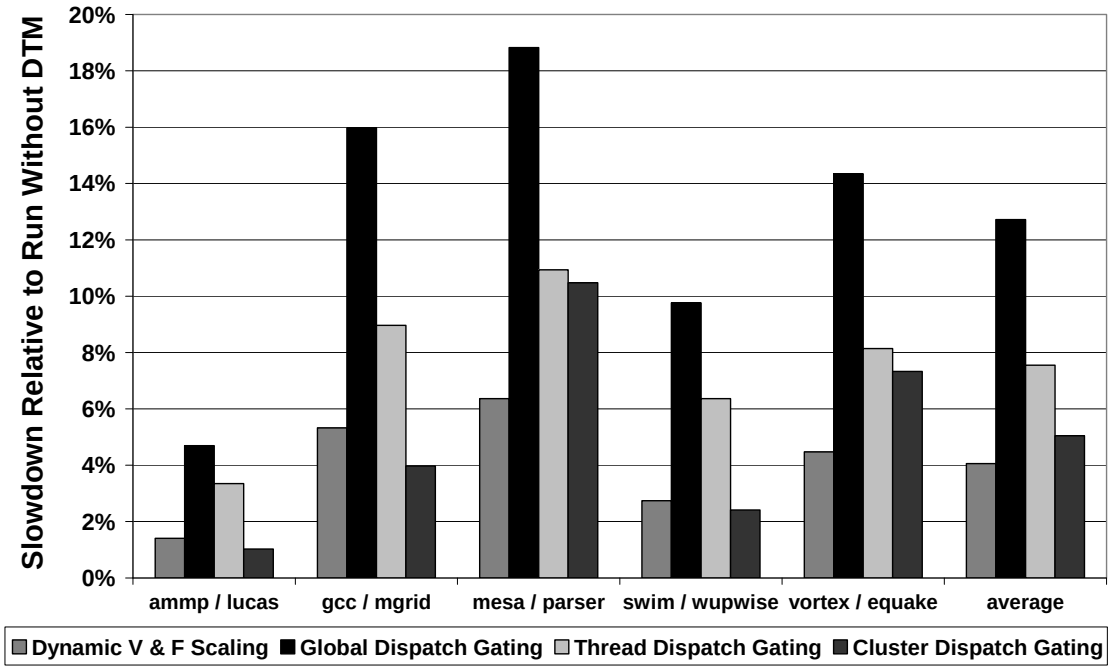


Figure 3.7: Performance of the dispatch gating policies on the clustered SMT design.

3.4.2. Heat Spreading Policies

Heat spreading policies exploit the different heating patterns of applications to reduce the frequency of thermal crises. By altering the assignment of threads to back-end clusters, these mechanisms cool a hot component of the chip by sending it instructions from a thread that does not heavily utilize that resource. For example, if a processor has a floating point application that overheats the FP units on its two clusters, steering an integer application to those clusters would cool the FP units while they are inactive. The goal of this approach is to keep all the back-end clusters active

all the time in contrast to policies based on activity migration which require idling resources [28][29][47][51][82][126].

Static heat spreading policies rearrange the assignment of threads to clusters at fixed intervals regardless of the application thermal behavior. These techniques are simple to implement and do not even use temperature measurements (except to determine when to engage dispatch gating – see below). Since static heat spreading methods do not respond to application behavior, they may potentially miss opportunities to cool threads more effectively. *Dynamic heat spreading* policies react to thermal conditions on the die by steering instructions or threads that are causing excess heating to cooler areas of the chip. Simultaneously, cooler instructions or threads are steered to the hot components of the processor to allow these areas to avoid thermal emergencies.

Heat spreading policies are not guaranteed to alleviate thermal emergencies. It may turn out that the "cold" thread enters a phase where it becomes quite hot, preventing the hotspot from cooling down. To ensure that the emergency threshold temperature is not exceeded, all heat spreading methods must engage a form of dispatch gating as a last resort. The dispatch gating mechanism engages at the normal dispatch trigger threshold, which is the temperature at which the heat spreading mechanism is deemed to have failed to address the temperature emergency. Because of the superiority of Cluster Dispatch Gating, it serves as this fail-safe backup cooling mechanism for all our heat spreading policies.

3.4.2.1. Static Heat Spreading Policies

We considered two static heat spreading algorithms. The first, Round Robin Steering, simply shifts the clusters assigned to each thread by one in a counter-clockwise direction after a fixed interval length. For example, if a thread is running on clusters 0 and 1 and the interval length is one million cycles, after one million cycles it

will be steered to clusters 1 and 2, and after two million cycles it will be steered to clusters 2 and 3.

The second algorithm, called Static Thread Swapping, swaps the clusters assigned to one thread with the clusters assigned to another after each fixed interval length. With two threads, this means that the first thread is steered to clusters 0 and 1 for one interval, then to clusters 2 and 3 for another interval, and then to clusters 0 and 1. The other thread is steered to the alternate set of clusters. We simulated policies with interval lengths ranging from 10,000 cycles to 50 million cycles and found that one million cycle intervals delivered the best performance.

Combining static heat spreading methods with dispatch gating is very simple. Both techniques are simply run together, with the spreading mechanism changing the assignment of threads to clusters, and Cluster Dispatch Gating engaging when the dispatch trigger threshold is reached, and disengaging at the stop threshold. Figure 3.8 illustrates the workings of the two static heat spreading policies.

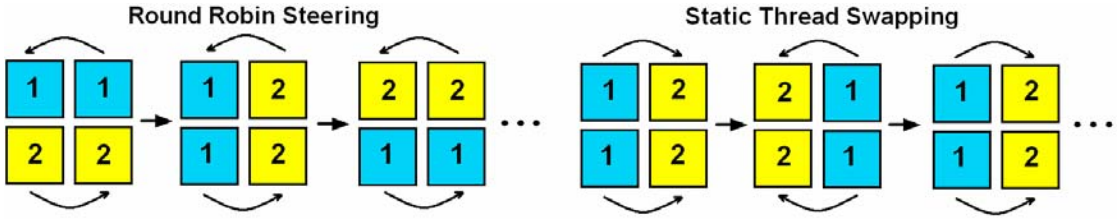


Figure 3.8: The two static heat spreading techniques.

3.4.2.2. Static Heat Spreading Results

The results for static spreading are shown in Figure 3.9 along with Cluster Dispatch Gating for comparison purposes. First of all, static heat spreading mechanisms definitely improve upon Cluster Dispatch Gating, decreasing the average slowdown from 5.0% to 1.3%. In particular, for the ammp/lucas workload, static heat spreading is so effective, that no dispatch gating is needed and the mechanisms had zero slowdown compared to the baseline. These spreading techniques also have a

performance degradation of less than a third of the slowdown of DVFS. While Round Robin Steering and Static Thread Swapping are about even in slowdown on non-uniform workloads, round robin is a slightly better mechanism across uniform workloads (not shown). The greater success of Round Robin was primarily due to its ability to more evenly spread threads among the clusters, since over the course of the run each thread is assigned to every pair of neighboring clusters. In Static Thread Swapping, threads either use clusters 0 and 1, or 2 and 3, and thus cooling is not as even.

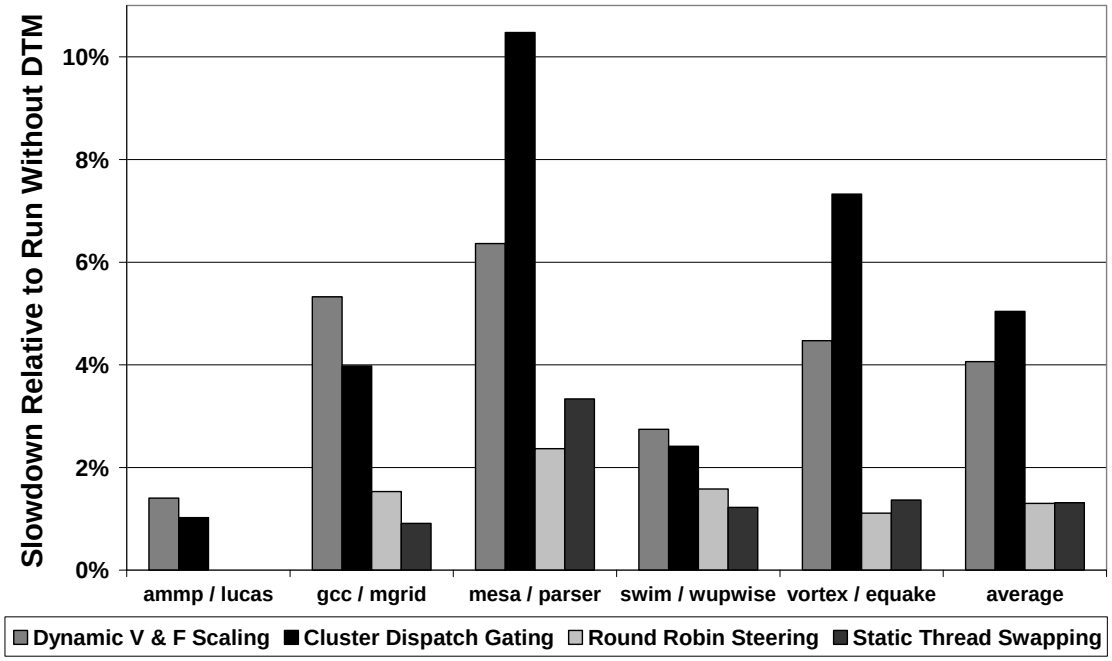


Figure 3.9: Performance of the static heat spreading policies on the clustered SMT design.

3.4.2.3. Dynamic Heat Spreading Policies

The dynamic heat spreading policies monitor the thermal behavior of the running applications and rearrange the cluster to thread assignment to balance the heating of the back ends. Every 10,000 cycles, the temperature sensors of the back-end clusters are examined. Then, dynamic spreading is applied to any back ends with a

peak temperature above the *spreading trigger threshold* (85.5°C), starting from the hottest cluster. To give the dynamic spreading techniques a chance to have an effect, the algorithm waits five million cycles before making another change to a back end involved in an earlier application of dynamic heat spreading. Furthermore, this prevents the algorithm from responding to small temperature fluctuations that do not threaten to cause thermal emergencies. Figure 3.10 provides a graphical explanation of the dynamic heating policies.

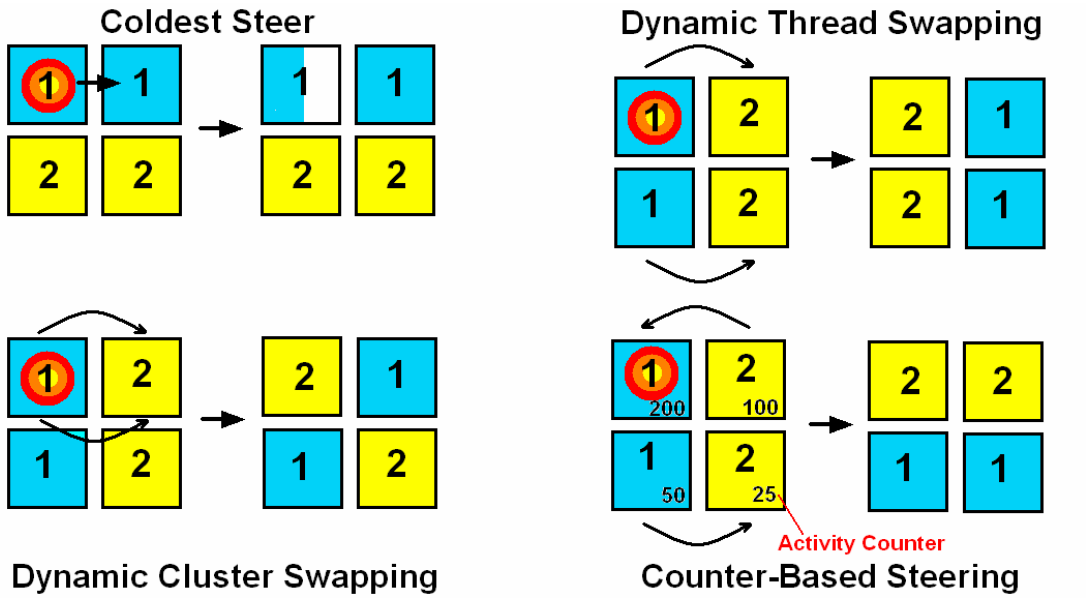


Figure 3.10: The four dynamic heat spreading techniques.

The first heat spreading mechanism, Coldest Steer, is an intra-thread spreading policy that works within a single thread's clusters. It is inspired by the T-Thermal algorithm of [28]. In order to avoid applying the policy unnecessarily and to account for noisy sensors, it requires that the difference in the peak temperatures between the hot and cold thread of a cluster be over a 0.2°C threshold before engaging. When engaged, it tries to send all instructions to the colder cluster of the two that are allocated to the thread. However, if the cold cluster cannot accept any more dispatched instructions because it has no free physical registers or issue queue slots, then

instructions are sent to the hot cluster if it has room. Ideally, this algorithm is successful in directing the bulk of the instructions to the colder cluster, leaving the hot cluster with a light load and a chance to cool off. The relevant thread can still suffer a performance drop because instructions are no longer steered according to the performance-conscious steering policy of the baseline architecture.

Dynamic Thread Swapping attempts to cool a hot thread by exchanging the clusters that the hot thread is using with the clusters assigned to the colder thread. Thus, it is a more sophisticated version of static thread-swapping policy. The colder thread is determined as the thread with the lowest peak temperature among all its clusters, and its instructions are steered to the hot clusters to permit them to cool. The hot thread is steered to the cooler clusters of the other thread, allowing the application to heat up this part of the back end without causing an immediate thermal emergency. Both the hot and the cold thread are steered to their new clusters for at least five million cycles. This prevents the cold thread from being falsely picked as a hot thread right after it was moved to the hot cluster and before the temperature has had a chance to change.

Often, the algorithm will swap two threads and after waiting five million cycles, it will find that the hot cluster is still hot. Before swapping the hot clusters again, the algorithm checks to see if that cluster has decreased in temperature. If the hot cluster is cooler than when it was swapped, the swapping is deemed to be successful and the thread running on the cluster is not swapped again, despite still being above the spreading threshold. This criterion prevents clusters that are slowly cooling down from being swapped prematurely. However, if the hot cluster was dispatch gated since last being swapped, then thread swapping proceeds normally, as the original swap clearly did not solve the heating crisis. These extra features were

experimentally found to improve the algorithm's performance over blindly swapping hot threads again after five million cycles.

The third dynamic spreading algorithm is Dynamic Cluster Swapping. This mechanism is very similar to Dynamic Thread Swapping. The only difference is that a single hot cluster can be individually swapped with a single cold cluster. Often the temperatures of the clusters associated with a given thread are quite different. Swapping at the granularity of a single cluster gives the spreading algorithm more flexibility to pick the best back ends to assign to a hot thread and does not force it to reassign clusters that are not causing thermal problems. On the other hand, cluster swapping may lead to configurations where a thread is using non-contiguous clusters in the back end, resulting in higher inter-cluster communication of register operand values.

Our final technique, Counter-Based Steering, employs activity counters for back-end components, in particular the number of instructions issued, the number of register accesses, and the number of times a functional unit is used. We examined the steering decisions made by the previous three inter-thread dynamic spreading techniques and found that often the algorithms make the mistake of swapping too often because they cannot determine whether the high temperature of a component is due to the activity of the current running thread or due to the previous thread that used it. This final algorithm addresses this problem by using activity counters as an indicator of a thread's thermal intensity as in [40][41][100]. These hardware counters are continuously active, tracking the usage of the various back-end components. When the algorithm detects a hotspot in a particular component, the activity counters for that component are checked and the thread which shows the least activity for that component is then steered to the cluster with the hotspot. The thread with the lowest activity is assumed to be the best choice for cooling that component. The thread is also

assigned to one of the neighboring clusters of the one with the hotspot (whichever neighbor is hotter) in order to keep the thread's assigned clusters contiguous. The thread originally assigned to the hot cluster is then steering to the other two remaining clusters. If the thread with the lowest activity in the hot component is already occupying that component's cluster, the algorithm chooses not to do anything, because moving threads will most likely just make the situation worse. Afterwards, the activity counters are reset while waiting for the next hotspot.

As with the static spreading policies, Cluster Dispatch Gating is engaged as a backup DTM mechanism for the dynamic spreading mechanisms if a hotspot reaches the dispatch trigger threshold, and is disabled at the stop threshold. The spreading trigger threshold is set to be the same value as the stop threshold for dispatch gating (85.5°C). This way, as soon as dispatch gating is turned off, spreading is engaged to further cool the hot thread and hopefully prevent the need to use dispatch gating in the future.

When Cluster Dispatch Gating is employed with Counter-Based Steering, there will be some interference between the two mechanisms. Since Counter-Based Steering observes the activity levels of back-end components, its readings will be affected by dispatch gating activity, which deactivates clusters for short periods of time. However, we observed that the interference was mostly constructive as clusters which were dispatch gated and thus had lower levels of activity also had lower temperatures, making them good choices to receive hot threads from the steering algorithm to alleviate hot clusters.

3.4.2.4. Dynamic Heat Spreading Results

Figure 3.11 shows the dynamic heat spreading results. Clearly Coldest Steer is ineffective, with overall performance worse than Cluster Dispatch Gating alone. There are three reasons for Coldest Steer's ineffectiveness in an SMT environment: (1) the

two clusters of the hot thread often have almost the same peak temperature, (2) the limited free resources in the cold cluster means that most of the instructions from the hot cluster cannot be moved, and (3) Coldest Steer prevents the very effective performance-driven steering mechanism from dispatching instructions to minimize communication between clusters and balance the instruction load.

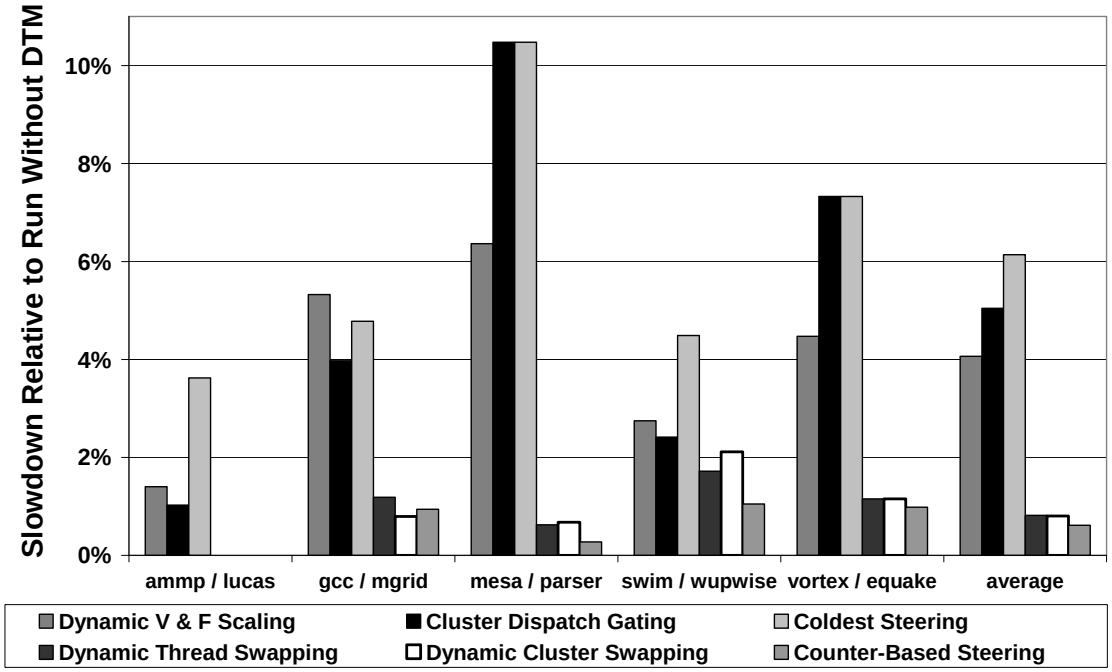


Figure 3.11: Performance of the dynamic heat spreading policies on the clustered SMT design.

On the other hand, our other three dynamic techniques are very effective at providing low cost DTM. Dynamic Thread Swapping and Dynamic Cluster Swapping both show a 0.8% slowdown on average. On individual workloads, sometimes thread swapping is superior and on other benchmarks cluster swapping is preferable. When it is most important to keep each thread's clusters together to decrease the communication costs of passing register values, thread swapping performs better. On other workloads, cluster swapping takes advantage of differences in the thermal

behavior of individual clusters to provide more precise targeting of hotspots and thus requires less use of dispatch gating.

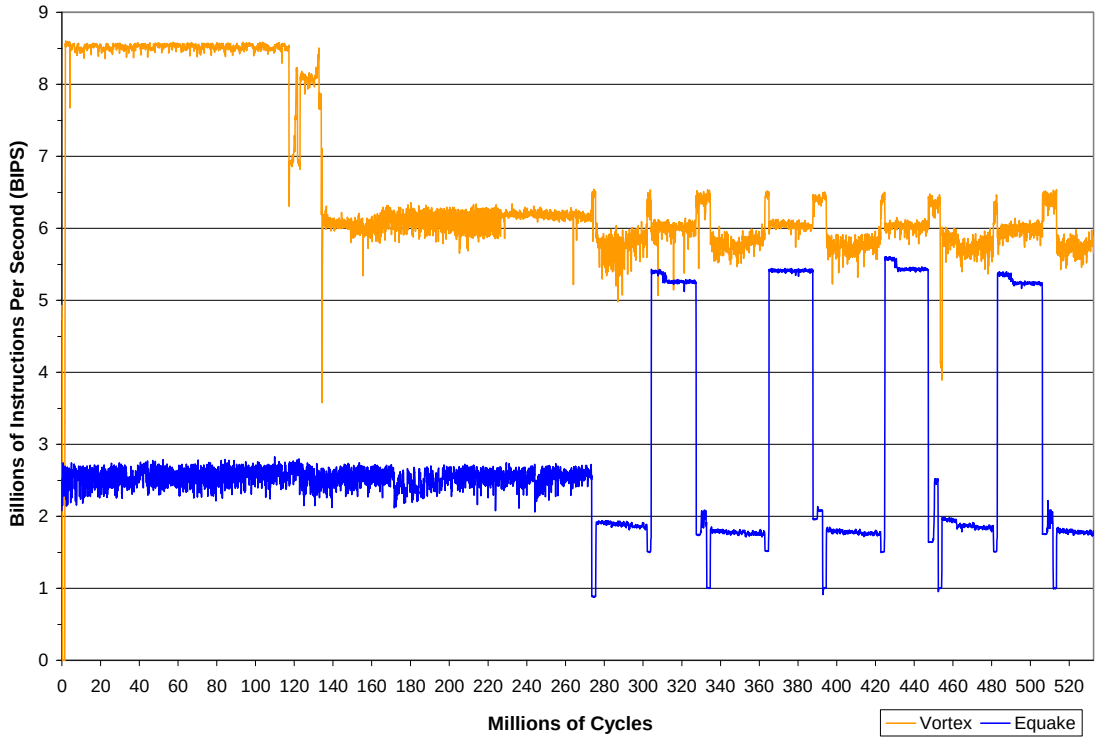


Figure 3.12: vortex / equake – performance with the Counter-Based Steering policy.

Finally, we find that the Counter-Based Steering policy performs best overall with a worst case degradation of only 1.0%, compared to 6.4% for DVFS. To gain a deeper understanding of the workings of Counter-Based Steering, Figure 3.12 shows the performance of the *vortex*/*equake* workload and Figure 3.13 shows the temperatures of the hottest resources during the simulation. Note the similarity of Figure 3.12 (Counter-Based Steering) with Figure 3.1 (no DTM), indicating little performance loss is incurred. Indeed, the performance loss on this benchmark pair is only 1.0% compared with 4.5% for DVFS (Figure 3.2). Undoubtedly, the Counter-Based Steering policy is better suited than DVFS to managing hotspots in a non-uniform workload and Figure 3.13 indicates why. By switching the thread to cluster assignment intelligently, Counter-Based Steering significantly evens out the

temperatures across all the clusters and eliminates most of the need for dispatch gating. Cluster Dispatch Gating is called as a backup failsafe just once, compared to the 27 times that DVFS is required. When DVFS is engaged, the frequency and voltage of the processor are decreased, reducing the performance of both benchmarks, as shown in Figure 3.2 by the multiple drops in the BIPS rate.

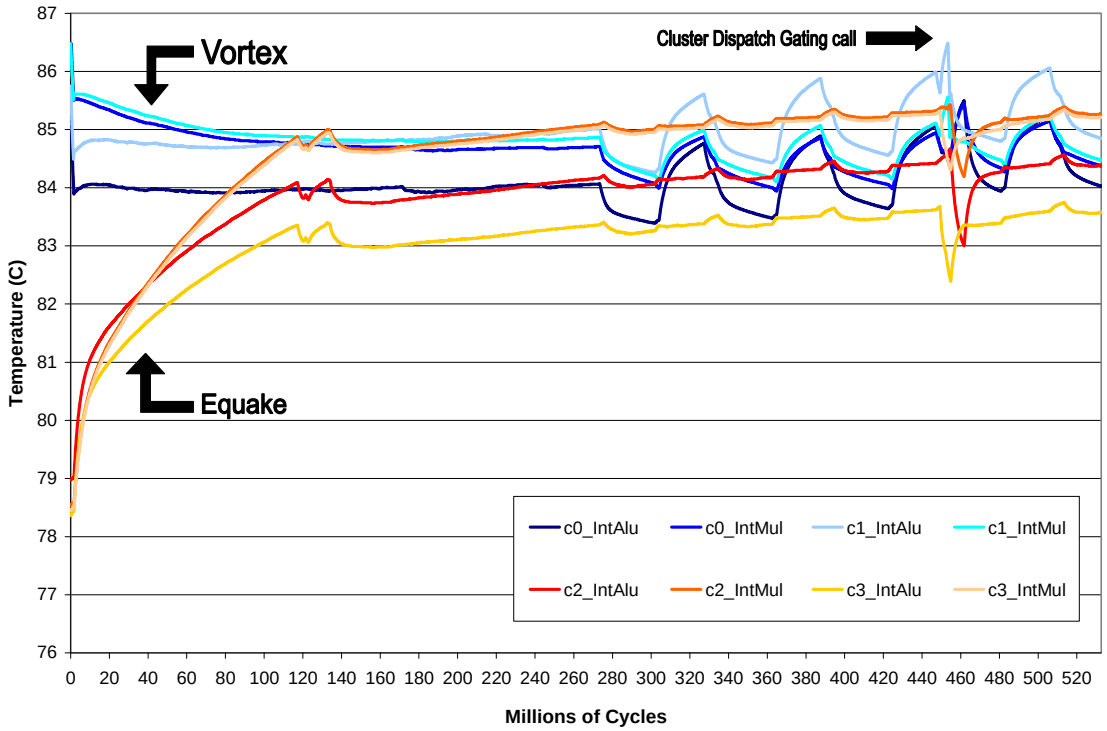


Figure 3.13: vortex / equake – temperature with the Counter-Based Steering policy.

Overall, the best steering-based DTM policies are Round Robin and Counter-Based Steering. While Round Robin Steering is not quite as effective as Counter-Based Steering, it is a very simple static policy to implement. Counter-Based Steering is appealing because of its particularly good performance on non-uniform workloads, where it demonstrates a significant advantage over DVFS. We therefore consider combining it with DVFS to achieve a "best of both worlds" DTM technique.

3.5. Combined Steering and DVFS Policies

Given the good match of dynamic voltage and frequency scaling to uniform workloads, and steering-based DTM's proficiency on non-uniform benchmark pairs such as *vortex*/*equake*, it seems natural to combine the two approaches. Thus, we considered a hybrid mechanism where Counter-Based Steering is used to spread the heat and dynamic voltage and frequency scaling takes the place of Cluster Dispatch Gating as the fail-safe mechanism for preventing a thermal crisis. (We implemented other hybrid policies using Round Robin and Dynamic Cluster Swapping, but found Counter-Based Steering to perform best.) Ideally, this combined policy will harness the temperature steering mechanism to obviate much of the need to use DVFS on non-uniform workloads, while we will get the performance benefits of DVFS on the uniform workloads where there is little temperature variation among clusters to exploit. Employing global DVFS as a backup mechanism does not cause interference with Counter-Based Steering. Reducing the frequency of the back ends changes the counter values read each interval in an absolute sense but does not affect the relative differences between component activities that are used to determine which threads are thermally intensive and need to be assigned to cold clusters. While it is possible to combine DVFS, Cluster Dispatch Gating, and Counter-Based Steering, we felt that merging three DTM mechanisms would be too complex to implement in hardware.

Figure 3.14 compares dynamic voltage and frequency scaling, Counter-Based Steering with Cluster Dispatch Gating (CDG), and the hybrid policy (Counter-Based Steering + DVFS) for both uniform and non-uniform workloads. As expected, DVFS is superior for uniform workloads, with an overall slowdown of 4.6% compared to 6.4% for Counter-Based Steering + CDG. Under these uniform conditions when both threads are hot simultaneously, it is difficult to beat the almost cubic power reduction of DVFS. However, due to the superiority of Counter-Based Steering + CDG on non-

uniform workloads, it provides better performance averaged across all workloads (uniform and non-uniform combined) with a degradation of 3.5%, compared to 4.3% with DVFS.

Looking at the hybrid policy, the last bar in Figure 3.14, we see that this mechanism is effective at merging the better performance features of DVFS and Counter-Based Steering. Specifically, this combination provides strictly better performance than DVFS, with degradations on uniform, non-uniform, and overall workloads at 4.2%, 1.4%, and 2.8% respectively (compared to 4.6%, 4.1%, and 4.3% for DVFS). Yet, it is still capable of obtaining most of the heat spreading benefit on non-uniform workloads. Furthermore, our results assume that the processor will see an equal mix of uniform and non-uniform workloads. In the event of an imbalanced workload of all uniform or all non-uniform thread combinations, Counter-Based Steering with DVFS provides near optimal performance regardless of the application mix, unlike either technique alone.

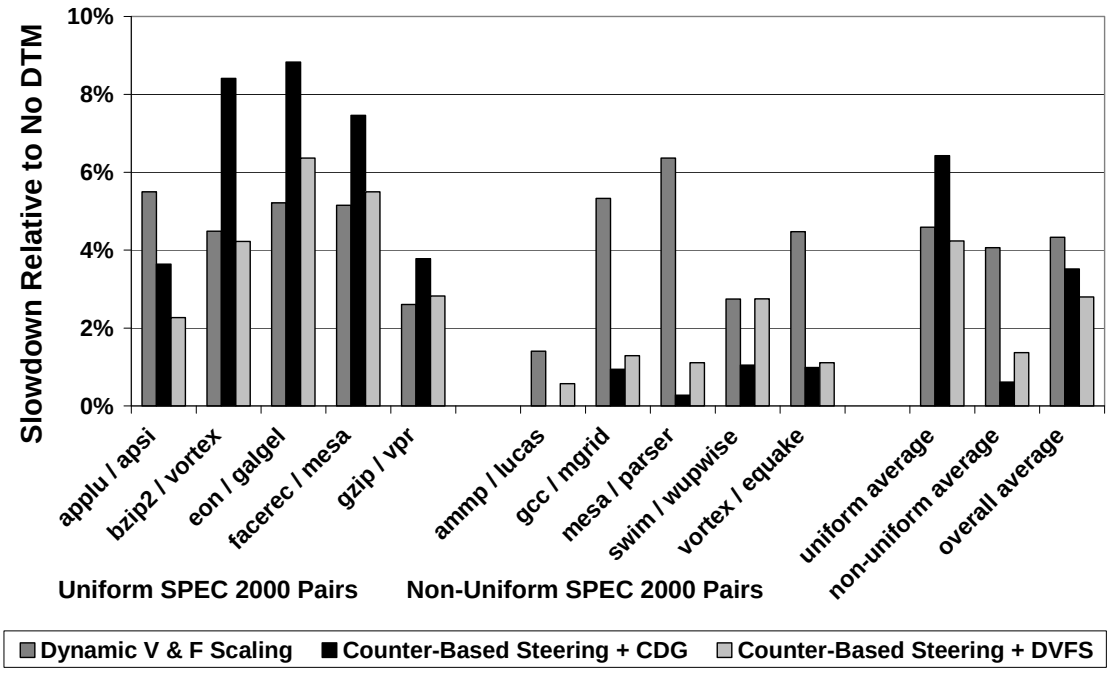


Figure 3.14: Overall performance comparison of the DTM techniques on the clustered SMT design.

3.6. Conclusions

In this chapter, we identify a class of SMT workloads where there is significant room to improve on the performance of dynamic voltage and frequency scaling. For non-uniform workloads with mixed integer and floating point applications or different thermal behaviors among the threads, DVFS's global effect cools one thread at the performance expense of the other. To address this deficiency, we propose DTM policies that leverage the built-in partitioning, steering, and thread migration mechanisms of clustered SMT architectures to provide effective temperature control with low implementation complexity. On non-uniform workloads, our best policy, Counter-Based Steering, provides effective DTM with only a 1% worst-case slowdown compared to a 6.4% slowdown for DVFS over a baseline without thermal management. Furthermore, our steering-based policy is competitive with DVFS across all workloads.

Our clustered SMT DTM policies have the additional advantage of not requiring the capability to scale frequency and voltage on a per-core level. This makes them very attractive for implementation in future large-scale chip multiprocessors where having numerous voltage domains will be undesirable, and possibly infeasible. Furthermore, our steering-based DTM techniques will function effectively in future technologies with ultra-low supply voltages which may pose problems for dynamic voltage and frequency scaling [27].

In order to take advantage of the complementary features of steering-based DTM and DVFS, we propose combining counter-based steering with dynamic voltage and frequency scaling. In this "best of both worlds" policy, Counter-Based Steering addresses non-uniform workloads by spreading the heat and minimizing the need to use DVFS, while uniformly hot benchmark pairs are cooled most effectively by dynamic voltage and frequency scaling. As a result, this hybrid policy provides the

best overall performance with a slowdown of 2.8% compared to 4.3% for DVFS and 3.5% for Counter-Based Steering with Cluster Dispatch Gating.

CHAPTER 4

APPLICATION SCHEDULING ALGORITHMS FOR UNPREDICTABLY HETEROGENEOUS CMP ARCHITECTURES

4.1. Introduction

As Moore's Law continues to deliver exponentially more transistors on a die over time, computer architects have transitioned to the multi-core approach whereby these transistors are used to create additional cores on the same die. Unfortunately, as transistors and wires shrink in every technology generation, they are becoming more susceptible to a variety of reliability problems. While transient (soft) errors are a near-term research focus, permanent (hard) errors and circuit variability are projected to become a significant challenge [13][14]. In this work, we concentrate on permanent faults and variations caused by imperfections in chip manufacturing and lifetime wear-out. These defects manifest as inoperable transistors, open or shorted wires, slower critical timing paths (decreasing operating frequency), and higher leakage power. Moreover, variations create weakened components which more easily wear out when subjected to the stress of high levels of activity, power, and temperature.

Ultimately, a major consequence of decreasing hardware reliability is that many cores on the die will have damaged components which will reduce their processing capabilities, increase their power dissipation, or even leave them inoperable. Manufacturers will not have the option of shipping only fully functional chips as this will necessitate unaffordably low yields. Instead, in order to provide reasonable performance at acceptable cost, future CMPs will have to be designed to tolerate faults and variations and operate in a degraded state [122]. Many researchers have developed resiliency techniques that adjust core frequency and voltage and deconfigure broken components, allowing processors to remain functional despite reliability problems [1][6][15][17][18][19][69][78][79][94][116][122][123][129].

However, these functional but degraded cores may still fail to provide the minimum expected level of performance and power efficiency throughout their expected lifetime.

In this chapter, we seek to mitigate the impact that the degradations and deconfigurations will have on the operation of future chip multiprocessors. Since hard errors and variations are largely the result of random physical processes that occur during manufacturing and usage, each core on a CMP will likely be uniquely affected. Thus, the resulting system will be an *unpredictably heterogeneous* chip multiprocessor (UH-CMP), even though it may have been designed as a homogeneous multi-core architecture. Figure 4.1 illustrates the concept of unpredictable heterogeneity by showing an initially homogeneous eight-core CMP that has incurred a number of hard faults, (represented by X's), creating heterogeneity among the cores due to these faulty components being de-configured.

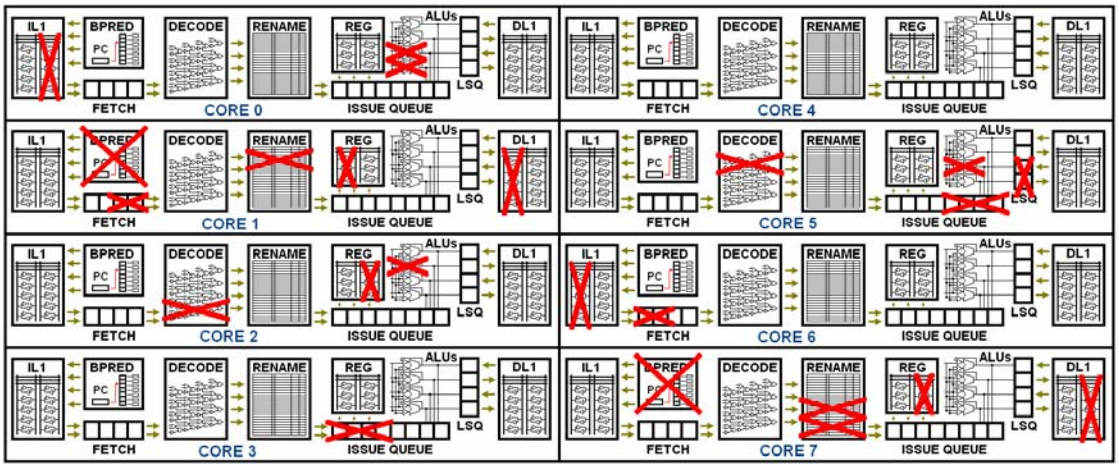


Figure 4.1: An illustrative example of an eight-core unpredictably heterogeneous chip multiprocessor.

Future software will consist of a set of applications with a variety of computational needs. We propose to exploit this workload diversity though self-tuning operating system scheduling algorithms that use high-level system feedback to match

application characteristics to core functionality in order to make the effects of the hard errors and variability imperceptible to the user.

We develop a combined hardware/software approach to address the complexity of the scheduling problem. The hardware is best capable of providing feedback on the performance and power dissipation of individual applications running on each of the degraded cores. On the other hand, the operating system is best situated to make the scheduling assignments, because it has a global perspective of the CMP and can balance the needs of the workload.

We explore two methodologies for attacking the scheduling problem. First, by assuming that application behavior changes slowly and that interactions between running threads are limited, we reduce the scheduling problem to the *Assignment Problem*, which can be solved by employing the Hungarian Algorithm [25][93]. Our second approach is to evaluate iterative optimization search algorithms that have been employed on many similarly difficult combinatorial problems [106][122]. These algorithms are simple to implement, have low computational requirements, and yet are extremely effective in practice. To our knowledge, our study is the first work to apply iterative optimization algorithms to heterogeneous multi-core thread scheduling.

4.2. Scheduling Algorithms for Unpredictably Heterogeneous CMPs

We propose scheduling algorithms that assign applications to cores over a fixed, relatively short period of time. Scheduling decisions are periodically reassessed to account for large application phase changes, programs completing, and new applications arriving to be processed. Our best algorithms consist of an exploration phase where samples of thread behavior on different cores are observed and a steady phase during which the algorithm runs the best schedule it found during the sampling phase. See Figure 4.2 for an example of this two-phase scheduling approach. In this example, there are 25 intervals in the exploration phase, each of four million cycles,

followed by a 900 million cycle steady phase, for a total scheduling period of one billion cycles.

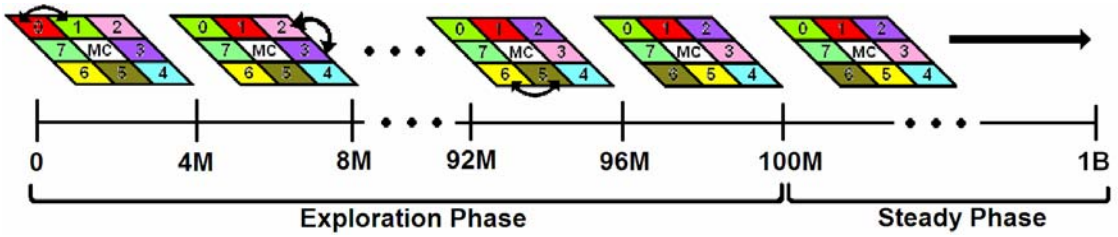


Figure 4.2: An example of the two-phase scheduling approach.

Table 4.1 compares the scheduling algorithms that we explore in this work. The table specifies the complexity of each algorithm where *n* is the number of cores (and the number of applications). For comparison purposes, we also implement Randomized and Round Robin Scheduling, two simple algorithms that have worked well on past multi-core designs.

Table 4.1: Scheduling algorithm summary.

Algorithm	**Exploration Phase (in cycles)**	**Complexity**
Randomized	none	O(n)
Round Robin	none	O(n)
Hungarian	8 intervals of 12.5M	$O(n^3)$ or $O(n^4)$ – See Section 4.2.2
Global Search	25 intervals of 4M	O(n)
Local Search	25 intervals of 4M	O(n)

4.2.1. Baseline Scheduling Algorithms

For the simple Round Robin and Randomized Scheduling Algorithms, we modeled 10 million cycle operating system time slices, the equivalent of 2.5 milliseconds. These algorithms do not require exploration and instead they use each time slice interval to perform their reassignments. The Round Robin Scheduler, illustrated in Figure 4.3, rotates the applications on cores so that each program runs on each degraded processor an even amount of time. This approach avoids a worst case assignment by limiting how long an application runs on any given core. The equal

assignment of applications to processors also avoids high power density scenarios and uneven wear-out of a core through over-activity or high temperature. Round Robin Scheduling has been found to be effective on statically designed heterogeneous CMPs and thus serves as a logical baseline [9].

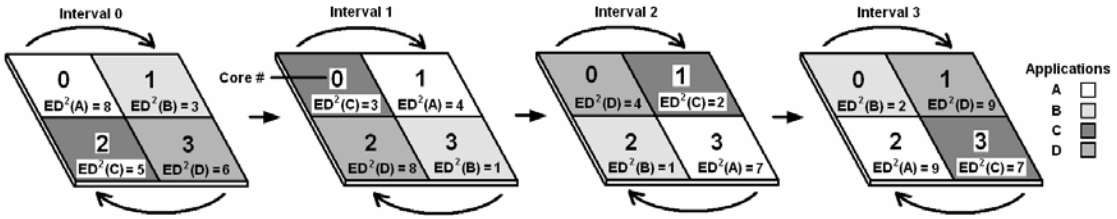


Figure 4.3: The Round Robin Scheduling Algorithm.

The Randomized Scheduling Algorithm, portrayed in Figure 4.4, simply picks a new arbitrary assignment each time slice interval. This scheduler will also evenly assign applications to cores in the long run and additionally avoids degenerate behavior that might occur with Round Robin's periodic cycling, such as destructive interference between the assignments and program phases.

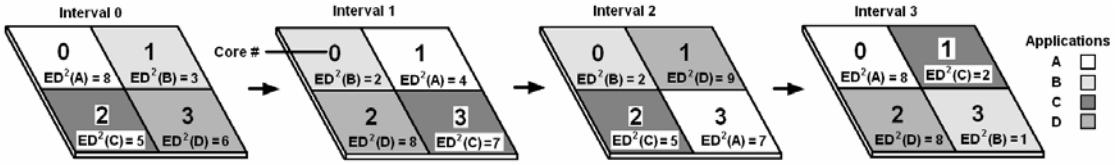


Figure 4.4: The Randomized Scheduling Algorithm.

4.2.2. Hungarian Scheduling Algorithm

The Hungarian Scheduling Algorithm is based on the Hungarian Algorithm developed by mathematicians to solve the well-known Assignment Problem, also called Weighted Bipartite Matching in graph theory [25][93]. During the exploration phase, the algorithm samples application performance and power statistics on each core and then picks the best scheduling assignment. In general, finding the best schedule is extremely difficult because threads interact during execution through contention for I/O and memory bandwidth as well as through heat conductivity

between cores. Furthermore, program behavior is dynamic both in the short-term time frame and over large program phases, such that sample information may not reflect future behavior.

In order to simplify the problem, our algorithm assumes that there are no such interactions among threads and that program behavior is static – at least for the duration of the scheduling period. Making these assumptions eliminates the interdependence between execution samples running simultaneously, reducing the scheduling problem to the Assignment Problem.

The Assignment Problem is defined as follows. Given an n×n cost matrix where the (i,j) element represents the cost of running application i on core j, find the assignment of applications to cores with lowest total cost. In our case, the elements of the cost matrix consist of the normalized energy-delay-squared (ED^2) product obtained by first sampling the execution of applications on each core. For each application, we divide each ED^2 sample by the ED^2 obtained during the first sampling interval to obtain the normalized values. Normalization ensures that applications are treated fairly by the scheduler despite any differences in the absolute value of their performance and power data.

Figure 4.5 outlines the six steps of Munkres' version [93] of the Hungarian Algorithm as described in [97]. The algorithm takes the cost matrix as input and proceeds by manipulating rows and columns through addition and subtraction to find a set of *starred* zero elements that represent the optimal assignment. During the algorithm, rows and columns are *covered* and zeroes are *starred* and *primed* to indicate special status. When the algorithm completes, there are *n* starred zeroes. A starred zero at location (i,j) means that the optimal solution to the Assignment Problem schedules application i to run on core j. The Hungarian Scheduler then uses the best assignment for the simplified problem as the schedule for the steady-state

phase. The Munkres version [93][97] of the Hungarian Algorithm is actually an $O(n^4)$ algorithm, not the optimal $O(n^3)$. In Chapter 7, we assess the impact of this difference in computational complexity on the algorithms' runtimes as the number of cores is scaled up.

Step 1: For each row of the matrix, find the smallest element and subtract it from every element in its row. Go to Step 2.

Step 2: Find a zero (Z) in the resulting matrix. If there is no starred zero in its row or column, star Z. Repeat for each zero in the matrix. Go to Step 3.

Step 3: Cover each column containing a starred zero. If *n* columns are covered, the starred zeros describe a complete set of unique assignments and the algorithm is done. Otherwise, go to Step 4.

Step 4: Find a non-covered zero and prime it. If there is no starred zero in the row containing this primed zero, go to Step 5. Otherwise, cover this row and uncover the column containing the starred zero. Continue in this manner until all zeros are covered. Save the smallest uncovered value and go to Step 6.

Step 5: Construct a series of alternating primed and starred zeros as follows. Let Z_0 represent the uncovered primed zero found in Step 4. Let Z_1 denote the starred zero in the column of Z_0 (if any). Let Z_2 denote the primed zero in the row of Z_1 (there will always be one). Continue until the series terminates at a primed zero that has no starred zero in its column. Unstar each starred zero of the series, star each primed zero of the series, erase all primes, and uncover every line in the matrix. Return to Step 3.

Step 6: Add the value found in Step 4 to every element of each covered row, and subtract it from every element of each uncovered column. Return to Step 4 without altering any stars, primes, or covered lines.

Figure 4.5: Munkres' six step Hungarian Algorithm [93][97].

4.2.3. Iterative Optimization Search Algorithms

Our other approach is to use iterative optimization search algorithms inspired by artificial intelligence research [106][112]. These iterative techniques operate with little domain-specific knowledge, are easy to implement, and have low computational complexity, making them highly suited to this complicated scheduling task. Furthermore, they continuously improve their solution as they execute, permitting the user to trade off algorithm runtime and solution quality as needed in his or her

implementation. The simplest search algorithms are greedy: they avoid searching in directions that initially appear to have performance slowdowns or power inefficiencies even if they may hold promise in the future. Therefore, these greedy algorithms may get stuck in local minima. However, in practice, greedy algorithms are quite effective in certain problem domains and are often used due to their simplicity. In this chapter, we study Global Search and Local Search.

In Global Search (Figure 4.6), the processor is configured into a new random schedule in each interval of the exploration phase of the algorithm. The operating system keeps track of the best configuration thus far and employs this configuration during the longer, steady-state phase. Figure 4.6 illustrates how Global Search operates on an example four core chip multiprocessor. Global Search has the advantage of rapidly exploring a broad range of configurations in a large search space such as a CMP with many cores. However, it may not arrive at a near-optimal solution.

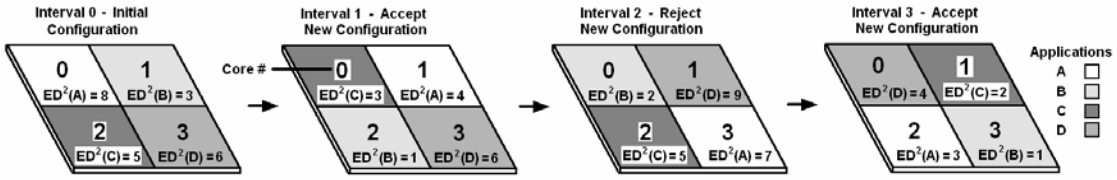


Figure 4.6: The Global Search Algorithm.

Local Search defines a neighborhood of assignment options that are closely related to the current configuration. During each exploration interval, a member in the neighborhood of the current assignment is selected as the next assignment. If this new assignment performs better than the original, then it is kept and Local Search proceeds from this new configuration. If the new assignment does not function as well as the original, then Local Search reverts to searching further in the neighborhood of the original solution. We define the neighborhood of a scheduling configuration as all

schedules that can be derived from the original schedule through some fixed number of pair-wise swaps.

In our results, we explore how many pair-wise swaps the algorithm should make per interval to determine the best setting. The advantage of selecting among a neighborhood of configurations that are derived from a few or even just one swap is that assignments in close proximity to the original are likely to have quite similar performance. This leads to a more gradual search that steadily improves the solution and avoids the large changes which could lead to poor results. On the other hand, increased swapping more rapidly explores the search space of assignments. Figure 4.7 demonstrates how Local Search works when one swap is performed per iteration. Figure 4.8 shows a two swap version of Local Search and highlights a key improvement in our algorithm which allows some of the swaps from an interval to be retained while others are discarded. We also implemented a version of Local Search that uses hill climbing [112] to escape local minima. We found, however, that the improvements over greedy search were minimal, indicating that the algorithms were not greatly impacted by local minima.

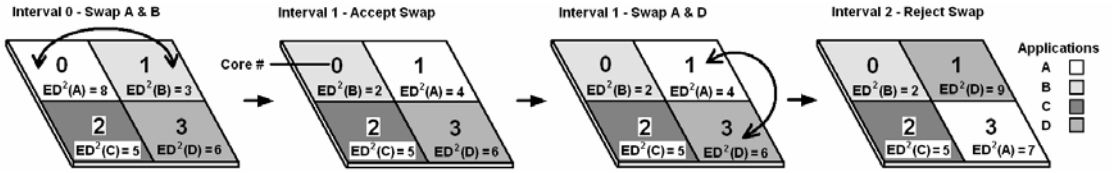


Figure 4.7: The one swap Local Search Algorithm.

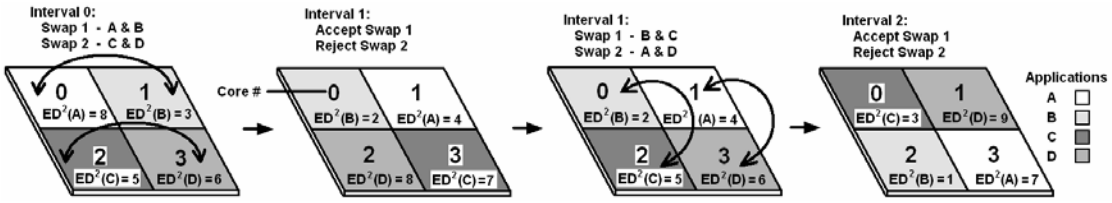


Figure 4.8: The two swap Local Search Algorithm.

4.3. Methodology

In this chapter and for the rest of the dissertation, our simulation infrastructure is based on the SESC simulator [107]. We improved the power and thermal modeling by augmenting SESC with Cacti 4.0 [132], an improved version of Wattch [23], the block model of Hotspot 3.0 [126], and an improved version of HotLeakage [142]. We extended Wattch and HotLeakage to model the dynamic and static power of all the units not addressed in Cacti 4.0, including logic structures such as the decoder, dependency check logic, issue queue selection logic, and ALUs. We assume a nominal clock frequency of 4.0 GHz and a supply voltage of 1.0V.

In order to efficiently simulate large multi-core architectures, we developed a parallel simulation framework. For this study, we focus on workloads of single-threaded applications chosen from the SPEC CPU2000 benchmarks. Multithreaded workloads will present a unique set of additional challenges when run on a heterogeneous CMP and we leave this added dimension to future work.

With these workloads, direct interaction among applications executing on different cores is limited. We assume that the on-chip cache hierarchy consists of private L1 and L2 caches, and thus do not have cache contention. While heat from one core conducted across the silicon die can cause inter-core heating effects, in our design, private L2 caches surround each core. These large caches have low and relatively uniform activity and thus act as heat sinks preventing much of the heating from another core from affecting its neighbors. The second major interaction among cores is their contention for off-chip memory bandwidth. We assume the bandwidth is statically partitioned among the cores. This avoids further complicating our already large search space of thread scheduling and core configuration options.

With these assumptions, we model a multi-core processor using a hierarchical simulation infrastructure. The two levels in our simulator and their interactions are

shown in Figure 4.9. This framework is reused in Chapters 5, 6, and 7 and is a critical enabling component of our experimental studies. The lower level shows the individual benchmarks being run in separate SESC simulations to obtain performance and power statistics. The top level is the chip-wide simulator which is implemented in Perl and performs two roles. First, it compiles the results from each core's simulation into the complete statistics for the whole chip multiprocessor. Second, it performs the role of the operating system by implementing the various scheduling algorithms and managing the execution of the single-core simulations for the exploration and steady phases of the simulation. (Figure 4.9 shows the top level performing global power management, which is not performed in this chapter but is used in Chapters 5 and 7.) The approach of running each benchmark separately and in parallel makes the experiment runtime relatively insensitive to the number of cores in the CMP, thus making our infrastructure highly scalable for studying future many-core architectures.

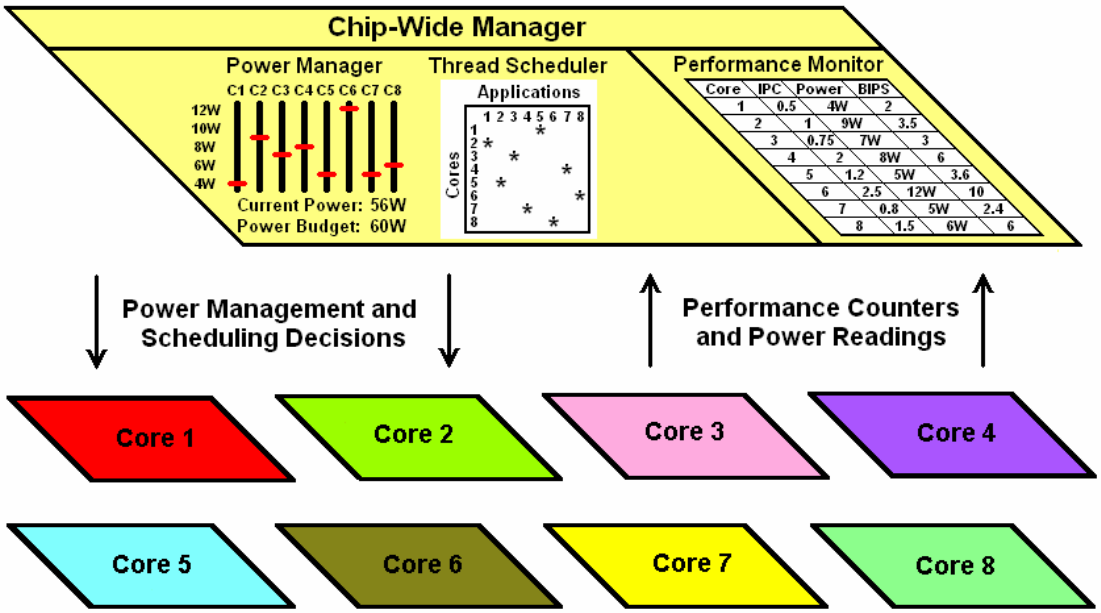


Figure 4.9: The hierarchical and parallel multi-core simulation framework.

Our baseline architecture consists of an eight-core homogeneous chip multiprocessor with no degradation due to hard failures or variations. Each core is a

single-threaded, 3-way superscalar, out-of-order processor. The main architectural parameters are listed in Table 4.2. In order to model temperature-dependent leakage power, we created a core floor plan. Each core is surrounded by its L2 cache modeled as four banks and illustrated in Figure 4.10.

Table 4.2: Core architectural parameters.

Front-End Parameters	
Branch Predictor	Hybrid of gshare and bimodal with 4K entries in the bimodal, gshare 2nd level, and meta predictor
Branch Target Buffer	512 entries, 4-way associative
Return Address Stack	64 entries, fully associative
Front-End Width	3-way
Fetch Queue Size	18 entries
Re-Order Buffer	100 entries
Retire Width	3-way
Back-End Parameters	
Integer Issue Queue	32 entries, 2-way issue
Integer Register File	80 registers
Integer Execution Units	2 ALUs and 1 multiply/divide unit
Floating Point Issue Queue	24 entries, 1-way issue
Floating Point Register File	80 registers
Floating Point Execution Units	1 adder and 1 multiply/divide unit
Memory Hierarchy	
L1 Instruction Cache	8KB, 2-way associative., 1 port, 1 cycle latency
Instruction TLB	32 entry, fully associative, 1 port
Load Queue	32 entries, 2 ports
Store Queue	16 entries, 2 ports
L1 Data Cache	8KB, 2-way associative, 2 ports, 1 cycle latency
Data TLB	32 entry, fully associative, 2 ports
L2 Cache	1MB, 8-way associative, 1 port, 10 cycle latency
Main Memory	1 port, 200 cycle latency

The task of modeling faults and variations in an architectural simulation is quite challenging. Much of the impact from errors and variability on a chip is highly device and circuit dependent and such low level details are not available at the time of initial architectural design. In this work, we focus on the architecturally visible effects of faults and variations. We study processor configurations that have become degraded

from the nominal design through manufacturing inconsistencies as well as wear-out over the lifetime of the device. For this study, the specific source of the degradation – manufacturing or wear-out – is not important because we focus on adapting the OS application scheduling to the core configuration *ex post facto*.

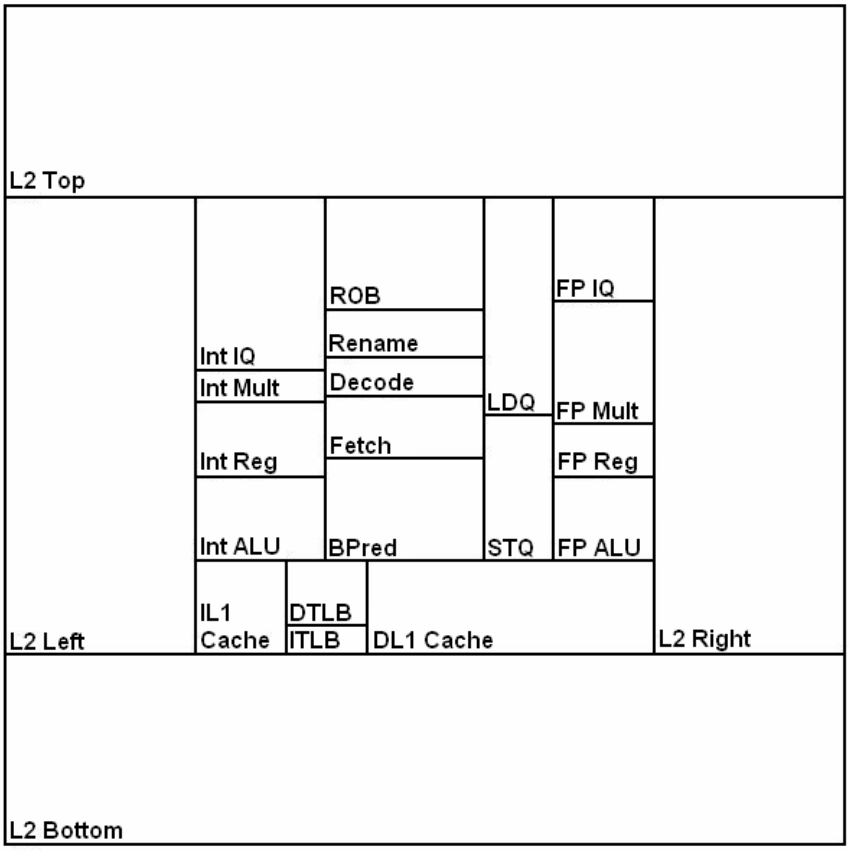


Figure 4.10: Processor core floor plan.

We focus on three forms of processor degradation. First, we model errors that cause the system to disable part of a pipeline component such as an ALU, load queue port, or set of ROB entries. We focus on large granularity errors that damage significant portions of the structure. Prior work has shown that when only a few entries in structures, such as an issue queue or register file, are damaged, the performance impact (assuming graceful degradation) is negligible, and thus adaptation is unnecessary [77]. Second, we assume core frequency degradation from

manufacturing process variations that result in slow transistors in critical circuit paths [77][94]. Prior work has found that these variations can increase processor cycle time by as much as 30%, eliminating an entire technology generation's worth of frequency improvement [13]. Likewise, using the VARIUS variation model [113], researchers found that cores on the same die could experience 20-50% differences in frequency with an average of 33% [134]. Third, we assume leakage current variations, which are also caused by process variations that diminish the quality of the transistors, magnifying sub-threshold and gate leakage currents.

Past research concluded that excessive leakage currents will be a very serious problem, with some [13] saying that leakage variability across dies could be as high as 20X. Others [55] suggest that, even at 45nm, within-die variations alone could cause leakage differences among cores of as much as 45%. Similarly, [134] found total power between cores to vary by 40-70% with an average of 53%. Following the arguments of [55][56], we focus on leakage variations that can be attributed to systematic variability. Thus, we consider leakage variations that affect an entire core as in [55] as well as those that affect a group of architectural blocks in close proximity.

Table 4.3: The degraded CMP configuration.

Core	Structural Faults	Frequency Degradation	Leakage Increase
1	2x normal memory latency (100 ns)	–	2x in the L1 caches
2	half the nominal size integer issue queue (16)	20% (3.2 GHz)	2x for the whole core
3	half the nominal size load queue (16)	10% (3.6 GHz)	2x in the store and load queues
4	one integer ALU is disabled	20% (3.2 GHz)	–
5	integer issue queue can only issue one instruction per cycle	–	–
6	half the L2 cache is broken leaving 500KB	10% (3.6 GHz)	2x in the integer cluster
7	half the nominal ROB entries (50)	–	2x in the floating point cluster
8	half the nominal size store queue (8)	–	2x in the front end

In a CMP where cores could be affected in a multitude of ways, there are numerous heterogeneous core configurations that could arise. In this study, we assume the degraded CMP configuration shown in Table 4.3. We assumed each core experienced some form of faults or variation but each processor was only affected by at most a few problems.

To test the effectiveness of our scheduling algorithms, we created the four eight-threaded workloads of SPEC CPU2000 applications shown in Table 4.4. Each benchmark was used evenly among the four workloads. For each simulation, we fast forwarded every benchmark five billion instructions, and then executed one billion cycles in SESC, or 0.25 seconds at a nominal frequency of 4 GHz. Cores that run at lower frequencies execute for proportionally fewer cycles.

Table 4.4: Application workloads.

Workload 1	applu, bzip2, equake, gcc, mcf, mesa, parser, swim
Workload 2	ammp, apsi, art, crafty, twolf, vortex, vpr, wupwise
Workload 3	mesa, ammp, applu, crafty, vortex, gcc, wupwise, mcf
Workload 4	swim, parser, vpr, bzip2, art, apsi, twolf, equake

The OS scheduler periodically switches between the exploration and steady-state phases of the algorithm. During the exploration phase, which constitutes 10% of the total execution time, the algorithm adapts to workload changes to find the best assignment of threads to cores. During the longer steady-state phase, the CMP runs with this best configuration. The performance of the algorithm is based on both the exploration and steady-state phases. The length and number of the sampling intervals are algorithm dependent parameters and are chosen to the best advantage of each technique. For each workload, we performed five different runs with different application-to-core starting assignments, and report the average, best, and worst results.

4.4. Results and Discussion

In this section, we present the results of the various scheduling algorithms on our degraded eight-core CMP. All comparisons are made using the energy-delay squared (ED^2) metric against a baseline with no errors or variations and an oracle scheduler which uses *a priori* knowledge to derive the best schedule among all possible options. We chose ED^2 as the metric in order to balance performance with power dissipation [83]. Section 4.4.1 discusses how the baseline schedulers compare to the non-degraded baseline. Section 4.4.2 shows how the Hungarian and AI search algorithms fare against the offline oracle. Finally, Section 4.4.3 provides an overall comparison of the scheduling algorithms.

4.4.1. Baseline Scheduling Algorithms

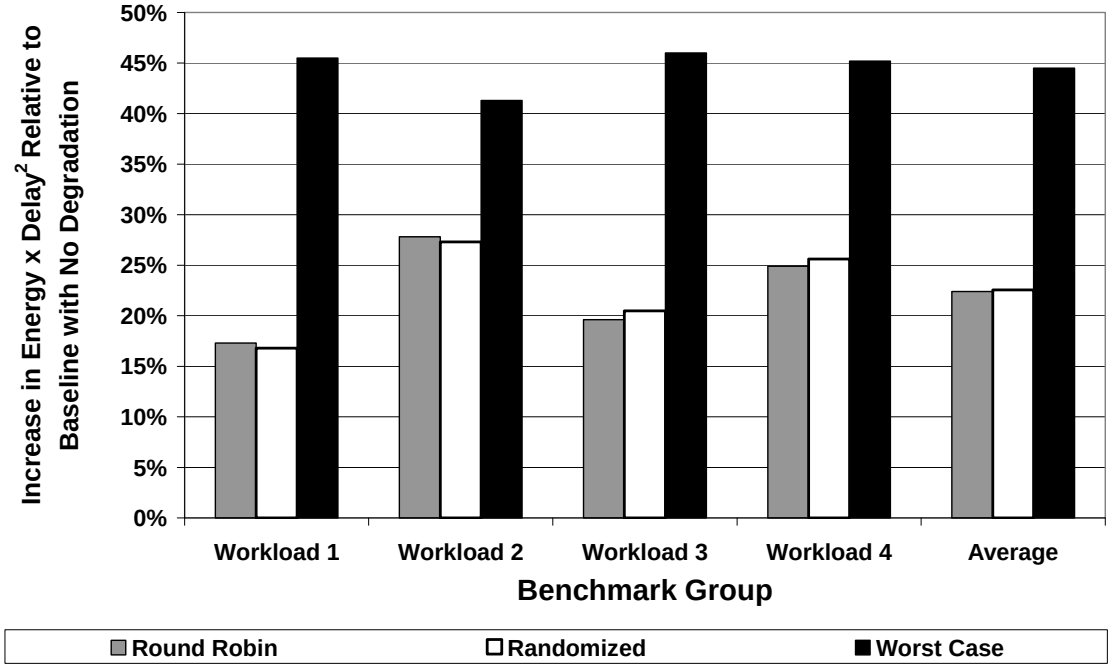


Figure 4.11: Comparison of the baseline scheduling algorithms.

We first evaluate the effectiveness of two simple scheduling algorithms, Round Robin and Randomized, on the degraded CMP of Table 4.3. Figure 4.11 shows the results of these schedulers on the degraded CMP relative to a baseline with no

degradation. Both approaches degrade ED^2 by over 22% on average. The final bar on the graph, the worst-case schedule, shows that an arbitrary assignment of threads to cores can degrade ED^2 by almost 45% compared to the baseline. Clearly, naïve policies can result in an unacceptable loss in power-performance that may render the degraded microprocessor unusable.

4.4.2. Hungarian Scheduling and Search Algorithms

The Hungarian Scheduling Algorithm samples each benchmark on each core during the exploration phase, and then computes the best assignment among all permutations (assuming no interactions or phase behavior). For the Hungarian Scheduler, the exploration phase is divided into eight intervals, each 12.5 million cycles long, during which the eight applications are executed once on each core, by starting with an initial assignment and then rotating the threads in a round robin fashion seven times. This allows the scheduler to generate the 8×8 cost matrix of ED^2 values to use as input to the algorithm.

Figure 4.12 shows the ED^2 of the Hungarian Scheduling Algorithm compared to the oracle scheduler. The solid bar represents the average of the five runs, and the range bars show the best (bottom) and worst (top) results. The algorithm performs well, suffering only a 7.3% increase in ED^2 relative to the oracle. The performance and power characteristics of the benchmarks during the initial 100 million cycle exploration phase are quite reflective of the overall traits of the benchmarks. Thus, using the Hungarian Algorithm to calculate the best solution among all possible scheduling permutations based on this sampling information yields a good assignment over the whole run, regardless of the starting assignment.

While effective, the Hungarian Scheduling Algorithm has $O(n^3)$ complexity, while the other algorithms are of O(n). We simulated the Hungarian Algorithm on our baseline core configuration and found it takes approximately 200K cycles to solve a

cost matrix with eight cores, a non-trivial cost that may not scale well to larger-scale CMPs. Since the number of sampling intervals scales linearly with the number of cores, a large amount of online profiling will also be required for chips with tens or hundreds of cores. Moreover, the algorithm may not work well when there are significant interactions among applications or rapid phase changes. These scalability issues are discussed in detail in Chapters 6 and 7.

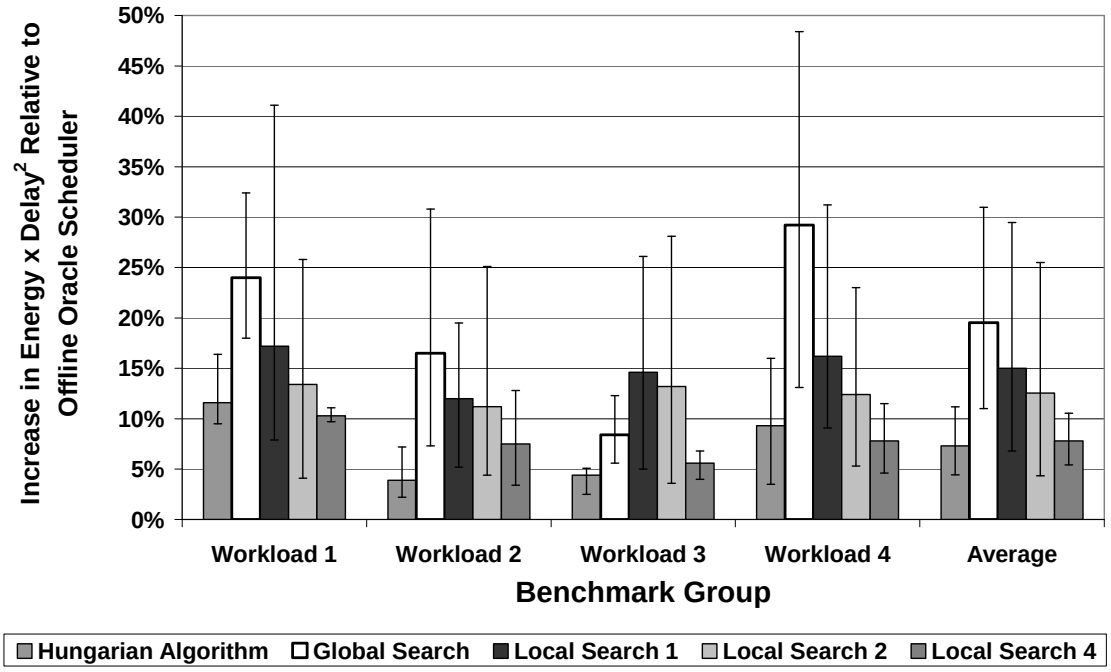


Figure 4.12: Comparison of advanced scheduling algorithms.

The Global and Local Search Algorithms divide the exploration phase into 25 intervals of four million cycles. Both start with the initial configuration and try other configurations, greedily pursuing paths that improve on the best schedule to date. Global Search simply tries the initial configuration and 24 other randomly chosen ones and then selects the best among them for the steady-state phase. This strategy sometimes works quite well but can perform poorly depending on the 25 configurations pursued. Overall, Global Search degrades ED^2 by 19.5% over the oracle scheduler.

Three versions of the Local Search method were implemented which vary in the number of pair-wise swaps performed to explore a neighboring configuration. Local Search N uses N pair-wise swaps such that two benchmarks are involved in each round for Local Search 1, while all benchmarks are swapped for Local Search 4. Local Search 1 makes a swap and then runs that schedule for the next four million cycle interval. If performance improves, it keeps that new configuration; otherwise, it selects another neighbor of the original solution. The comparison is made using the average of the normalized ED^2 (with respect to the ED^2 of the previous interval) of the two threads involved in the swap. Local Search 2 and Local Search 4 have an additional feature to improve their performance. Instead of collectively accepting or rejecting all the swaps made in an interval, beneficial pair-wise swaps are kept and others discarded. From the results in Figure 4.12, the additional pair-wise swaps of Local Search 2 and Local Search 4 significantly improves the algorithm; the ED^2 increase achieved with one, two, and four pair-wise swaps each interval is 15.0%, 12.6%, and 7.8%, respectively. Moreover, Local Search 4 significantly outperforms Global Search. The range bars show that Local Search 4 is also less sensitive to the initial assignment due to its ability to more rapidly search the space of possible assignments.

4.4.3. Overall Comparison

In Figure 4.13, we compare all the scheduling algorithms to the non-degraded chip multiprocessor. The offline oracle scheduler achieves 3.1% better ED^2 than the CMP without degradation. This occurs due to the fact that some of the degraded cores operate at lower power, due to lower frequency or failed components that are power gated. Consequently, an omniscient scheduler can find an assignment that is more power-performance efficient than the baseline.

Moreover, both the Hungarian and Local Search 4 Scheduling Algorithms achieve ED^2 values very close to the non-degraded baseline – higher only by 3.2% and 3.7%, respectively – compared to the over 22% degradations with naïve schedulers. Thus, intelligent scheduling will be critical to maintaining acceptable levels of power-performance efficiency on future CMPs degraded by faults and variations.

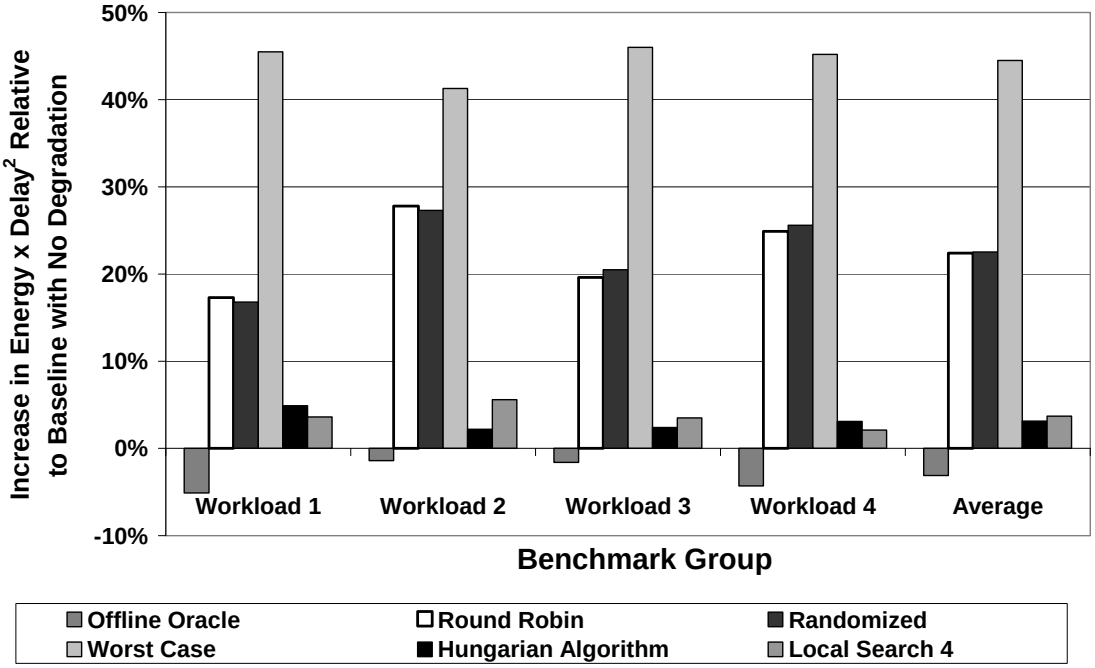


Figure 4.13: Overall comparison.

4.5. Conclusions

In future CMPs, hard errors and variability will conspire to create dynamic heterogeneity among the cores. Unlike static designed-heterogeneous chip multiprocessors, the unpredictability of manufacturing defects, wear-out mechanisms, and variations will require self-tuning scheduling techniques that efficiently find a near-optimal schedule given any degraded CMP scenario, thereby making the chip degradation imperceptible to the user. In this chapter, we devise a number of different scheduling algorithms for finding near-optimal thread to core assignments in a degraded CMP.

We first demonstrate that naïve policies, such as Round Robin and Randomized Scheduling, degrade ED^2 to the point that the chip may be rendered unusable. Under the assumption of limited core-to-core interaction, we observe that the scheduling problem reduces to the Assignment Problem and can be addressed through the Hungarian Algorithm. We devise a scheduler based on this algorithm that achieves an ED^2 close to that of an oracle scheduler. We further develop schedulers based on AI search techniques that obviate the requirement of limited core-to-core interaction, and that better scale to large CMP organizations. The most scalable and effective of these policies rapidly arrives at a near-optimal solution that degrades ED^2 by only 3.7% over a non-degraded architecture, compared to over 22% for prior scheduling approaches.

CHAPTER 5

GLOBAL POWER MANAGEMENT ALGORITHMS FOR UNPREDICTABLY HETEROGENEOUS CMP ARCHITECTURES

5.1. Introduction

In this chapter, we further investigate unpredictably heterogeneous chip multiprocessors, but shift our focus from application scheduling to global power management. In recent technology nodes, voltage scaling has failed to keep pace with increasing transistor densities, causing dynamic power to increase. Furthermore, leakage current has also been growing in successive technology generations, exacerbating the need for improved power management schemes [21][92].

As mentioned in the related work, most prior research has focused on energy efficiency in terms of metrics such as energy-delay-squared (ED^2) or energy-per-instruction (EPI) [37][43][52][62][71][72][75][115][118]. However, these efforts are more targeted towards battery life and power supply costs. In this chapter, we focus on the more recently proposed problem of keeping the chip-wide power dissipation under a fixed budget [10[53][58][114[119][134]. While localized methods of power management that ignore the big picture can be very effective for energy efficiency, meeting a budget requires global coordination and is thus a more challenging problem for large-scale chip multiprocessors. This challenge becomes even more complex with dynamic heterogeneity since offline or static power management methods fail to consider the unpredictable nature of the system. Most prior work [10][58][114][119] has only considered homogeneous CMPs, while two other efforts [53][134] have looked at global power management in chip multiprocessors suffering from process variations, but without manufacturing defects or wear-out faults. This chapter presents the first global power management algorithm targeted to address the high degree of uncertainty of unpredictably heterogeneous CMPs.

Our novel approach to power management is called the *Throughput-Aware Power Allocation Scheme (TAPAS)* and it is unique in that it focuses on controlling power directly, rather than extrapolating based on voltage and frequency settings. Consequently, our technique reduces the incidence of power overshoots as well as ensures the available power is best utilized, leading to higher performance. In addition, by shifting responsibility for managing dynamic voltage and frequency scaling (DVFS) to the individual cores, rather than managing DVFS at the global level as in [10][58][119][134], the algorithm is less dependent on analytical power-performance models (critical in prior techniques) that may become inaccurate as a result of the unpredictable and complex effects of hardware faults and process variations.

The rest of the chapter is organized as follows. The next section discusses the workings of our baseline power management techniques, the TAPAS method, and the associated Up/Down DVFS scheme. Section 5.3 explains our simulation methodology and Section 5.4 analyzes our results. There are many possible directions to extent this work and they are discussed in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.2. Global Power Management

In this chapter, we focus on studying hierarchical schemes for global power management where the goal is to maximize total performance for a given chip-wide power budget, as introduced in [58]. Our hierarchical schemes consist of two layers, a lower layer where each core operates independently, and a top layer global power manager (GPM) responsible for inter-core coordination. Each core can operate in a number of possible power levels which provide different power-performance tradeoffs. At the lower layer in the hierarchy, the cores are responsible for the local power management activities associated with their current power level and also supply the GPM with performance and power information on which to base its decisions. Performance information can be obtained from hardware counters available in most

modern processors. Power data can be derived from current sensors which can be embedded in the cores, such as those in the Foxton controller [84]. The global power manager is tasked with collecting and processing the cores' performance and power information and assigning power levels based on this data to each core to maximize throughput under the power budget.

The GPM could be implemented by the operating system, in a system hypervisor, in a microcontroller such as the Foxton controller, or as a separate thread running on one of the cores [58][134]. All of our power management policies use voltage and frequency scaling to throttle power dissipation in the cores and meet the power budget. We experimented with global and per-core fetch toggling policies but found them to be less effective and do not discuss them further in this chapter. The next section discusses the details of Chip-Wide DVFS and the MaxBIPS global power managers which are employed in most prior research [10][58][119][134] and serve as our baselines. In section 5.2.2, we describe TAPAS and highlight how it differs from prior GPM approaches. Section 5.2.3 discussed the Up/Down DVFS algorithm which runs on each core as the lower layer of the hierarchy in the TAPAS policy.

5.2.1. Baseline DVFS Algorithms

In prior research, the most common approach to global power management has been to task the global layer in the power manager with the responsibility of setting the voltage and frequencies of the CMP's cores to keep the total power consumption under the budget [10][58][119][134]. This is accomplished by employing analytical models which relate performance and power dissipation to voltage and frequency. Using these models, the GPM calculates a best setting of voltages and corresponding frequencies which is estimated to maximize total chip throughput while not exceeding the power threshold. In order to understand these models and the GPM algorithms, Figure 5.1 provides a list of definitions for the all the terminology employed.

n: The number of applications as well as cores in the system.
$A = \{a_1 \ldots a_n\}$: The set of applications in the system.
$C = \{c_1 \ldots c_n\}$: The set of cores in the system.
p : The number of power levels in the system. In our case p = 7.
$L = \{1 \ldots p\}$: The set of power levels in the system.
m : The middle power level, $m = \lceil p / 2 \rceil$. In our case m = 4.
$gpm(c_i) = l_j$: A function representing the global power management assignment mapping core c_i to power level l_j.
$VR = \{v_1, v_2, v_3, \ldots, v_p\}$: The range of voltage levels to which the cores can be set. These levels correspond to the power levels in L. For our algorithms, VR = {0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0}.
$v_{low} = v_1$: the lowest voltage allowable by DVFS.
$v_{high} = v_p$: the highest voltage allowable by DVFS.
$volt(l_i) = v_i$: A function mapping power level, l_i, to voltage setting, v_i. In our case, volt(1) = 0.7V, volt(2) = 0.75V, volt(3) = 0.8V, volt(4) = 0.85V, volt(5) = 0.9V, volt(6) = 0.95V, and volt(7) = 1.0V.
P_{max}: The chip-wide power budget.
$thr(a_i, c_i, l_j) = t_k$: A function representing the calculated throughput, t_k, of application a_i on core, c_i, at the power level, l_j. Throughput, t_k, is real-valued and is measured in the units billions of instructions per second (BIPS).
$sam\_thr(a_i, c_i, l_j) = t_k$: This function represents taking a sample of the throughput, t_k, of application, a_i, on core, c_i, at the power level l_j running in the chip multiprocessor. Throughput, t_k, is real-valued and is measured in the units billions of instructions per second (BIPS).
$est\_thr(a_i, c_i, l_j) = t_k$: A function representing the estimated throughput, t_k, of application, a_i, on core, c_i, at the power level l_j. Throughput, t_k, is real-valued and is measured in the units billions of instructions per second (BIPS).
$pow(a_i, c_i, l_j) = p_k$: A function representing the power, p_k, of application, a_i, on core, c_i, at the power level l_j. Power, p_k, is real-valued and is measured in watts.
$sam\_pow(a_i, c_i, l_j) = p_k$: This function represents taking a sample of the power of application a_i on core c_i at the power level l_j running in the chip multiprocessor. Power, p_k, is real-valued and is measured in watts.
$est\_pow(a_i, c_i, l_j) = p_k$: A function representing the estimated power, p_k, of application, a_i, on core, c_i, at the power level l_j. Power, p_k, is real-valued and is measured in watts.
$PR = \{pr_1, pr_2, pr_3, \ldots pr_p\}$: The range of power values (in watts) that can be allocated to the cores by the TAPAS global power management algorithm. These levels correspond to the power levels in L. In our experimental setup, there are 7 power levels and PR = {3, 4.5, 6, 7.5, 9, 10.5, 12}.
$powl(l_i) = pr_i$: A function mapping power level, l_i, to the corresponding value in the power range, pr_i. In our case, powl(1) = 3W, powl(2) = 4.5W, powl(3) = 6W, powl(4) = 7.5W, powl(5) = 9W, powl(6) = 10.5W, and powl(7) = 12W.
P_{avail}: The current amount of unallocated power available for the algorithm to allocate.
$alloc\_pow(c_i) = pr_j$: A function mapping core, c_i, to a power value, pr_i, in the range PR which has been allocated by the TAPAS global power management algorithm.

Figure 5.1: Global power management definitions.

In a multi-core processor with dynamic voltage and frequency scaling, there are numerous possible configurations of the core voltages and frequencies. Sampling all possible combinations would require a large amount of sampling intervals and lead to a significant amount of time where the CMP was running in a suboptimal configuration. Instead the GPM samples the cores at one voltage and frequency level and then uses the analytical models to estimate the performance and power that would result if the cores were set to other DVFS levels. In our unpredictably heterogeneous system, each core and its assigned application must be sampled because the relationship between the voltage and frequency setting and power-performance will vary for a given application/core pair according to the application's execution characteristics and the core's degradations.

To speed up and simplify these estimations, a number of assumptions listed in Figure 5.2 are used [58]. First, power is assumed to be proportional to frequency and to the square of the voltage (assumption 1), which is good approximation if static power is ignored, since for dynamic power, $P = CFV^2$. Together with assuming frequency is proportional to voltage (assumption 2), power is derived to have a cubic dependence on voltage (assumption 3). Then coupling an assumption that throughput is proportional to frequency (assumption 4) with the assumption 2, throughput can be estimated as proportional to voltage (assumption 5). From these assumptions, formulas are developed, shown in Figure 5.3 which calculate the expected throughput and power of an application/core pair at other power levels from the sampled throughput and power values [58].

$pow(a_i, c_i, l_j) \propto freq(c_i, l_j) \times volt(l_j)^2$	(1)
$freq(c_i, l_j) \propto volt(l_i)$	(2)
(1) & (2) $\rightarrow$ $pow(a_i, c_i, l_j) \propto volt(l_j)^3$	(3)
$thr(a_i, c_i, l_j) \propto freq(c_i, l_j)$	(4)
(2) & (4) $\rightarrow$ $thr(a_i, c_i, l_j) \propto volt(l_j)$	(5)

Figure 5.2: Assumptions in GPM power-performance modeling.

The task of finding the best power levels for the cores of the CMP can be transformed into an optimization problem, formulated in Figure 5.4. This optimization problem is discrete, since voltages are set to a set of fixed values, and nonlinear due to the cubic voltage/power relationship. We employ exhaustive search as in [10][58][119] to determine the best set of power levels for the cores to apply during the next power management interval. As pointed out in [58], one advantage of exhaustive search is that the constants c_i and k_i in Figure 5.4 do not have to have to be explicitly computed to run the GPM. Instead, the ratio of the voltages of the sampled and desired values is used to estimate the required throughput or power as shown in the formulas in Figure 5.3. However, exhaustive search is an idealized method of computing the best voltages and frequencies, and a more feasible approach would be to convert the problem to a continuous, linear optimization problem as in [134].

$$\text{est\_pow}(a_i, c_i, l_i) = \text{pow}(a_i, c_i, m) \times (\text{volt}(l_i) / \text{volt}(m))^3 \quad (1)$$
$$\text{est\_thr}(a_i, c_i, l_i) = \text{thr}(a_i, c_i, m) \times (\text{volt}(l_i) / \text{volt}(m)) \quad (2)$$
$$\text{est\_thr}(a_i, c_i, l_i) = \text{thr}(a_i, c_i, m) \times (\text{powl}(l_i) / \text{powl}(m))^{1/3} \quad (3)$$

Figure 5.3: Formulas for estimating power and throughput using model.

Maximize Throughput: $\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} c_i \cdot v_i$

Subject to the global power budget and voltage constraints:

$$\forall i: \quad v_{low} \le v_i \le v_{high}$$

$$\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} k_i \cdot v_i^3 \le P_{max}$$

c_i and k_i are constants specific to each application and core pair.

Figure 5.4: The traditional formulation of global power management as an optimization problem.

Linear optimization would have to compute the constants as well as linearize the cubic voltage terms in the global power budget constraint. Furthermore, since only certain discrete values can be assigned for voltage and frequency according to the available power levels, the closest discrete value to the linear programming (LP) solution would need to be selected. The need to be conservative in selecting discrete values (to avoid power overshoots) may lead to a sub-optimal selection of voltage

levels. Alternatively, an integer linear programming (ILP) solver could be used to find the best discrete values directly and avoid the need for conservative selection. While implementing LP and ILP was beyond the scope of this chapter, Chapter 7 explores these techniques and investigates their effectiveness and computational complexity.

Current voltage regulators for dynamic voltage and frequency scaling are off-chip, restricting the number of voltage domains possible on the chip [64]. Furthermore, off-chip regulators are limited by the speed that they can change chip voltages [64]. The limitation on DVFS switching rate is one reason the top layer GPM which operates only at OS context switch granularity, is given responsibility for setting voltage and frequency. Chip-Wide DVFS is a simple GPM which uses a single voltage and frequency for all cores, that represents the capabilities of a using a single off-chip regulator for the cores [58][64][119]. In this policy the v_i in Figure 5.4 must all be the same value, greatly reducing the search space of the optimization. The Chip-Wide DVFS algorithm must find the highest voltage level for the cores which still meets the global power budget, as this will provide the highest attainable performance.

Table 5.1: Power levels for chip-wide DVFS, per-core MaxBIPS DVFS, and TAPAS.

Power Mode	Voltage	Scaled Frequency for Given Nominal (Max) Frequency				TAPAS
		Max 4.0 GHz	Max 3.6 GHz	Max 3.2 GHz	Max 2.8 GHz	
7	1.00 V	4.0 GHz	3.60 GHz	3.20 GHz	2.80 GHz	12.0 W
6	0.95 V	3.8 GHz	3.42 GHz	3.04 GHz	2.66 GHz	10.5 W
5	0.90 V	3.6 GHz	3.24 GHz	2.88 GHz	2.52 GHz	9.0 W
4	0.85 V	3.4 GHz	3.06 GHz	2.72 GHz	2.38 GHz	7.5 W
3	0.80 V	3.2 GHz	2.88 GHz	2.56 GHz	2.24 GHz	6.0 W
2	0.75 V	3.0 GHz	2.70 GHz	2.40 GHz	2.10 GHz	4.5 W
1	0.70 V	2.8 GHz	2.52 GHz	2.24 GHz	1.96 GHz	3.0 W

In [58], only 3 voltage/frequency levels were employed giving each core only three power levels. However, [10][119] found that having more levels benefited the algorithms and Chip-Wide DVFS in particular. Thus, we experimented with using 3, 5, 7, and 11 levels and found 7 levels to provide almost all the benefit of 11 with

reduced implementation complexity. Both the Chip-Wide DVFS and per-core MaxBIPS use the 7 power levels described in Table 5.1. We assume global power management decisions must be made at a coarse granularity, to allow the OS to be involved and provide time for chip-wide coordination, and use an interval of 10 ms as in [134]. Rather than have specific sampling periods, the GPM for Chip-Wide DVFS and MaxBIPS simply uses the performance and power data observed over the previous OS interval as the sample to calculate the best voltages and frequencies for the cores during the next interval as in [58].

The pseudocode for Chip-Wide DVFS is given below:

Input:

(1) The number of cores and applications, n.

(2) A set of applications, A, assigned to corresponding cores, C.

(3) The power levels $l_1 \ldots l_n$ that each core was set to during the last interval.

(4) The function *volt()* mapping power levels, L, to the voltage range, VR.

(5) The global power budget P_{max}.

Output:

(1) The global power management assignment function *gpm()* assigning power levels, L, to cores, C, to maximize overall throughput.

(2) The total best estimated throughput, $T_{total,}$ of the best global power assignment.

(3) The total power dissipation, $P_{total,}$ for the best global power assignment.

Steps:

1. (*Sample the throughput and power of the applications and cores at their previous power level, which is l_i for core i*)

   For i from 1…n:

   $thr(a_i, c_i, l_i) = sam\_thr(a_i, c_i, l_i)$.

$pow(a_i, c_i, l_i) = sam\_pow(a_i, c_i, l_i)$.

End.

2. (*Iterate through the power modes to find the highest power mode each core can be set to that meets the global budget*)

   For i from p…1:

   a. $P_{total} = 0$.

   b. (*Estimate and sum the power of each core at this voltage level*)

      For j from 1…n:

      i. (*Calculate the power of core j using formula (5.3-1)*)

         $est\_pow(a_j, c_j, i) = pow(a_j, c_j, l_j) \times (volt(i) / volt(l_j))^3$

      ii. $P_{total} = P_{total} + est\_pow(a_j, c_j, i)$.

      End.

   c. (*Check if the total power is below the power budget. If so, set the cores to this power level, calculate the total chip throughput, and return*)

      If $P_{total} \leq P_{max}$

      $T_{total} = 0$.

      For j from 1…n:

      i. $gpm(c_j) = i$.

      ii. (*Calculate the throughput of core j using formula (5.3-2)*)

         $est\_thr(a_j, c_j, i) = thr(a_j, c_j, l_j) \times (volt(i) / volt(l_j))$.

      iii. $T_{total} = T_{total} + est\_thr(a_j, c_j, i)$.

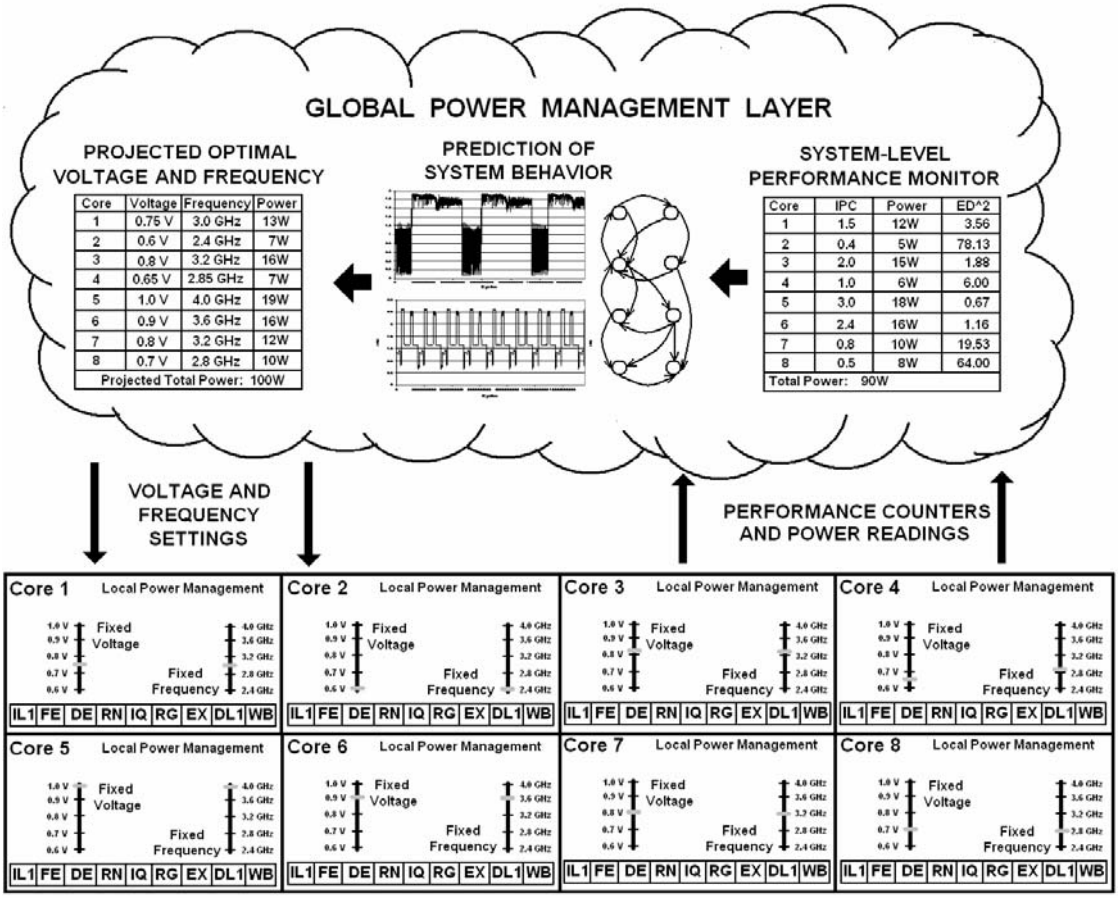
      End.

      Return($gpm()$, T_{total}, P_{total}).

      End.

   End.

We also compare TAPAS to the MaxBIPS GPM algorithm proposed by Isci et al. [58], which employs per-core DVFS. This policy requires a different power grid for each core, necessitating multiple on-chip voltage regulators for processors with many cores. Setting each core's voltage and frequency independently gives the global power manager much more freedom to extract as much performance out of the power budget as possible, but greatly increases the size of the search space. Figure 5.5 provides an illustration of a global power management system based on the MaxBIPS algorithm.



Core	Voltage	Frequency	Power
1	0.75 V	3.0 GHz	13W
2	0.6 V	2.4 GHz	7W
3	0.8 V	3.2 GHz	16W
4	0.65 V	2.85 GHz	7W
5	1.0 V	4.0 GHz	19W
6	0.9 V	3.6 GHz	16W
7	0.8 V	3.2 GHz	12W
8	0.7 V	2.8 GHz	10W
Projected Total Power: 100W			

Core	IPC	Power	ED^2
1	1.5	12W	3.56
2	0.4	5W	78.13
3	2.0	15W	1.88
4	1.0	6W	6.00
5	3.0	18W	0.67
6	2.4	16W	1.16
7	0.8	10W	19.53
8	0.5	8W	64.00
Total Power: 90W			

Figure 5.5: An overview of global power management using the MaxBIPS algorithm.

The pseudocode for the MaxBIPS algorithm is given below:

Input:

(1) The number of cores and applications, n.

(2) A set of applications, A, assigned to corresponding cores, C.

(3) The power levels $l_1 \ldots l_n$ that each core was set to during the last interval.

(4) The function *volt()* mapping power levels, L, to the voltage range, VR.

(5) The global power budget P_{max}.

Output:

(1) The global power management assignment function *gpm()* assigning power levels, L, to cores, C, to maximize overall throughput.

(2) The total best estimated throughput, $T_{best\text{-}total,}$ of the best global power assignment.

(3) The total power dissipation, $P_{best\text{-}total,}$ for the best global power assignment.

Steps:

1. (*Sample the throughput and power of the applications and cores at their previous power level, which is l_i for core i*)

   For i from 1…n:

   $thr(a_i, c_i, l_i) = sam\_thr(a_i, c_i, l_i)$.

   $pow(a_i, c_i, l_i) = sam\_pow(a_i, c_i, l_i)$.

   End.

2. (*Calculate the estimated total throughput and total power of every possible combination of power allocations for the CMP using brute force. Find the combination with the maximum total throughput that does not exceed the global power budget*)

   $T_{best\text{-}total} = 0$, $P_{best\text{-}total} = 0$.

   For j_1 from 1…p:

   (*Calculate the throughput of core 1 at power level j_1 using formula (5.3-2)*)

   $est\_thr(a_1, c_1, j_1) = thr(a_1, c_1, l_1) \times (volt(j_1) / volt(l_1))$

   (*Calculate the power of core 1 at power level j_1 using formula (5.3-1)*)

$est\_pow(a_1, c_1, j_1) = pow(a_1, c_1, l_1) \times (volt(j_1) / volt(l_1))^3$

For j_2 from 1…p:

(*Calculate the throughput of core 2 at power level j_2 using formula (5.3-2)*) $est\_thr(a_2, c_2, j_2) = thr(a_2, c_2, l_2) \times (volt(j_2) / volt(l_2))$

(*Calculate the power of core 2 at power level j_2 using formula (5.3-1)*) $est\_pow(a_2, c_2, j_2) = pow(a_2, c_2, l_2) \times (volt(j_2) / volt(l_2))^3$

For j_3 from 1…p:

(*Calculate the throughput of core 3 at power level j_3 using formula (5.3-2)*) $est\_thr(a_3, c_3, j_3) = thr(a_3, c_3, l_3) \times (volt(j_3) / volt(l_3))$

(*Calculate the power of core 3 at power level j_3 using formula (5.3-1)*) $est\_pow(a_3, c_3, j_3) = pow(a_3, c_3, l_3) \times (volt(j_3) / volt(l_3))^3$

⋮

For j_n from 1…p:

A. (*Calculate the throughput of core n at power level j_n using formula (5.3-2)*) $est\_thr(a_n, c_n, j_n) = thr(a_n, c_n, l_n) \times (volt(j_n) / volt(l_n))$

(*Calculate the power of core n at power level j_n using formula (5.3-1)*) $est\_pow(a_n, c_n, j_n) = pow(a_n, c_n, l_n) \times (volt(j_n) / volt(l_n))^3$

B. (*Sum up the total throughput and power for this combination of power level settings*) $T_{total} = 0$, $P_{total} = 0$.

For k from 1…n

$T_{total} = T_{total} + est\_thr(a_k, c_k, j_k)$.

$P_{total} = P_{total} + est\_pow(a_k, c_k, j_k)$.

End.

C. (*If the total power of this configuration is below budget and it has higher throughput than the current best, make it the best*)

If $P_{total} \leq P_{max}$ and $T_{total} > T_{best\text{-}total}$

I. $T_{best\text{-}total} = T_{total}$.

II. $P_{best\text{-}total} = P_{total}$.

III. (*Set the power level assignment to this new found best configuration*)

For k from 1…n

$gpm(c_k) = j_k$.

End.

End.

End.

End.

End.

End.

3. Return (gpm(), $T_{best\text{-}total}$, $P_{best\text{-}total}$).

In prior work [10][58][119][134], the global layer of the CMP power manager was responsible for setting the core voltage and frequency levels. Consequently, the rate of DVFS changes was limited by cross-core communication speeds, by the frequency at which a hypervisor or operating system routine could be executed, and by the runtime of the exhaustive MaxBIPS algorithm or linear optimization. In addition to enabling per-core voltage and frequency domains, new on-chip voltage regulator

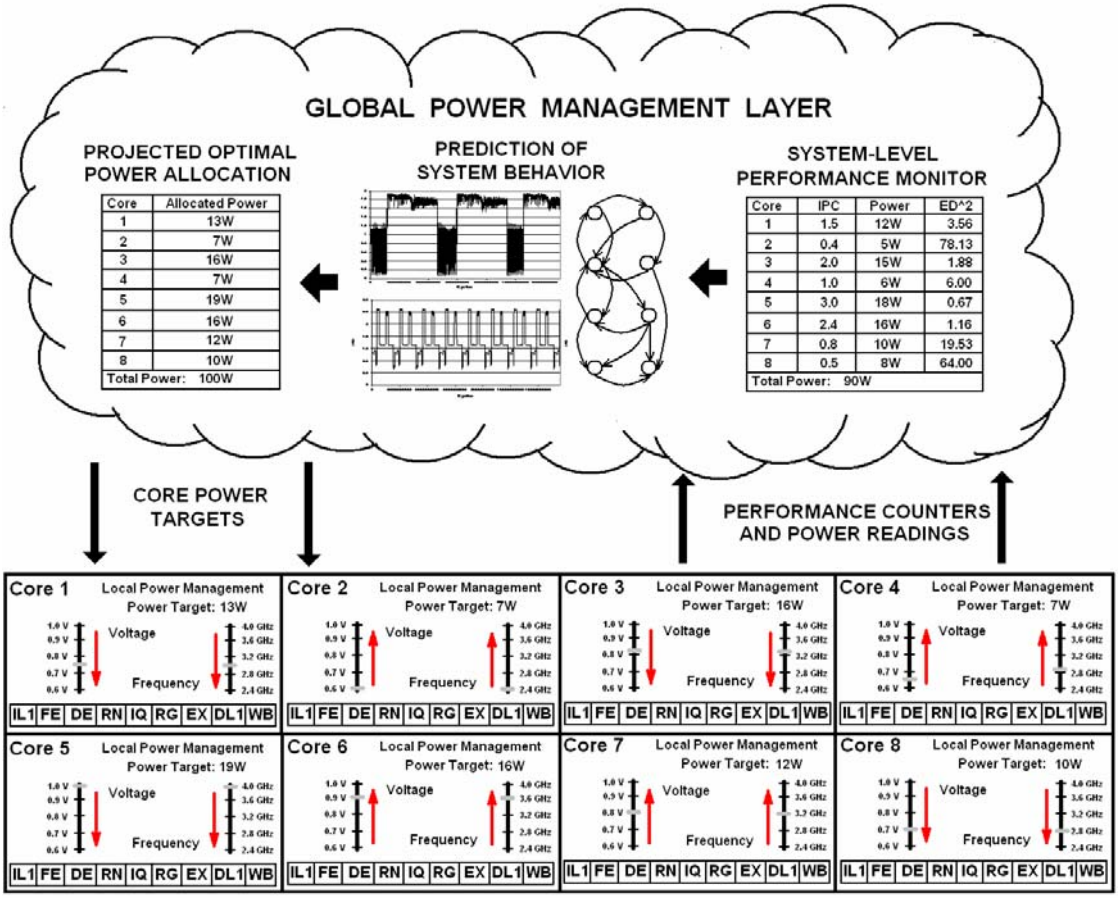
technology [64] will allow DVFS to be employed at much finer time granularities. However, with voltage and frequency scaling decisions made at the global layer as in these prior studies, it is not possible to take advantage of this new DVFS capability. In the next section, we discuss our novel TAPAS scheme which is designed to explicitly take advantage of new finer-grained DVFS hardware.

5.2.2. Throughput-Aware Power Allocation Scheme

We propose a new scheme which allows the power manager to take advantage of the rapid transition times of future on-chip voltage regulators. Assigning voltages and frequencies at coarse granularities such as 10 ms can potentially miss opportunities to maximize throughput and may also result in unnecessary violations. Variations in application behavior over the course of this large interval could lead to lower than anticipated activity levels, generating power at levels below those anticipated by the management algorithm, thus wasting opportunities to increase throughput by raising the frequency. On the other hand, these application phase changes could result in higher than expected activity causing larger than projected power, leading to a chip-wide power overshoot. The problem of projecting the power dissipation for a given voltage and frequency is further complicated by process variations which create another source of error in any analytical model [134].

To address the challenges of global power management in the presence of dynamic application behavior and unpredictably heterogeneous cores, we developed the Throughput-Aware Power Allocation Scheme (TAPAS). As in Chip-Wide DVFS and MaxBIPS, the TAPAS global power manager uses performance and power samples from the previous power management interval to calibrate its power-performance model for each core. Rather than assign a specific voltage and frequency to each core at the global layer, the GPM calculates a power target for each core which will maximize overall chip throughput. The cores are then individually responsible for

adjusting their voltage and frequency levels to stay as close to the target as possible. Leveraging new on-chip voltage regulator technology, these adjustments are made at a fine granularity according to the Up/Down DVFS mechanism described in the next section. Since the ultimate goal of global power management is to meet a specific power budget, it makes sense to directly allocate power targets to the cores instead of guessing which voltage and frequency levels the cores should be set to in an effort to reach the power budget. A diagram of the operation of TAPAS global power management is shown in Figure 5.6.



Core	Allocated Power
1	13W
2	7W
3	16W
4	7W
5	19W
6	16W
7	12W
8	10W
Total Power:	100W

Core	IPC	Power	ED^2
1	1.5	12W	3.56
2	0.4	5W	78.13
3	2.0	15W	1.88
4	1.0	6W	6.00
5	3.0	18W	0.67
6	2.4	16W	1.16
7	0.8	10W	19.53
8	0.5	8W	64.00
Total Power:	90W		

Figure 5.6: : An overview of global power management using the TAPAS algorithm.

Like Up/Down DVFS and MaxBIPS, the TAPAS power allocation problem can be formulated as an optimization problem, shown in Figure 5.7, which can be

considered the inverse of the formulation for MaxBIPS (Figure 5.4). Again, the assumptions of Figure 5.2 and the formulas of Figure 5.3 are used.

Maximize Throughput: $\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} c_i \cdot p_i^{1/3}$

Subject to the voltage constraints and global power budget:

$\forall i: v_{low} \leq v_i \leq v_{high} \Rightarrow k_i \cdot v_{low}^3 \leq p_i \leq k_i \cdot v_{high}^3$

$\sum_{i=1}^{n} p_i \leq P_{max}$

c_i and k_i are constants specific to the each application and core pair.

Figure 5.7: The TAPAS formulation of the global power management optimization problem.

Unlike the MaxBIPS equations where the independent variable controlled by the power management algorithm is the voltage, here the power management algorithm directly assigns power values. Since the power allocated to each core is not bound to discrete values as the voltage and frequency would be, our power allocation algorithm is more amenable to being solved by linear optimization and slower, more costly integer/discrete optimization is not needed. Linear optimization still requires the maximization function and the voltage constraints to be linearized as discussed further in Chapter 7.

However, for this study, we assume that the core power can only be set to one of seven discrete levels. This provides for a more direct comparison the original implementation of MaxBIPS [10][58][119]. Yet, it potentially limits TAPAS from utilizing its full potential, because given a continuous range of power options, the scheme could potentially find a better solution. We determine the median power level by dividing the total chip budget by the number of cores. Three lower power levels are allowed as well as three higher power levels as described in Table 5.1. The pseudocode for the TAPAS algorithm is given below:

Input:

(1) The number of cores and applications, n.

(2) A set of applications, A, assigned to corresponding cores, C.

(3) The power levels $l_1 \ldots l_n$ that each core was set to during the last interval.

(4) The function *powl()* mapping power levels, L, to the power allocation values in PR.

(5) The global power budget P_{max}.

Output:

(1) The global power management assignment function *gpm()* assigning power levels, L, to cores, C, to maximize overall throughput.

(2) The corresponding power allocation function *alloc_pow()* which apportions power from the range PR to cores, C, to maximize overall throughput.

(3) The total best estimated throughput, $T_{best\text{-}total,}$ of the best global power allocation assignment.

(4) The total power dissipation, $P_{best\text{-}total,}$ for the best global power allocation.

Steps:

1. (*Sample the throughput of the applications and cores at the power level, l_i, each core, c_i, was set to during the previous power management interval*)

   For i from 1…n:

   $thr(a_i, c_i, l_i) = sam\_thr(a_i, c_i, l_i)$.

   End.

2. (*Calculate the estimated total throughput and total power of every possible combination of power allocations for the CMP using brute force. Find the combination with the maximum total throughput that does not exceed the global power budget*)

   $T_{best\text{-}total} = 0$, $P_{best\text{-}total} = 0$.

   For j_1 from 1…p:

   (*Calculate the throughput of core 1 at power level j_1 using formula (5.3-3)*)

$est\_thr(a_1, c_1, j_1) = thr(a_1, c_1, l_1) \times (powl(j_1) / powl(l_1))^{1/3}$

For j_2 from 1…p:

(*Calculate the throughput of core 2 at power level j_2 using formula (5.3-3)*) $est\_thr(a_2, c_2, j_2) = thr(a_2, c_2, l_2) \times (powl(j_2) / powl(l_2))^{1/3}$

For j_3 from 1…p:

(*Calculate the throughput of core 3 at power level j_3 using formula (5.3-3)*)

$est\_thr(a_3, c_3, j_3) = thr(a_3, c_3, l_3) \times (powl(j_3) / powl(l_3))^{1/3}$

⋮

For j_n from 1…p:

A. (*Calculate the throughput of core n at power level j_n using formula (5.3-3)*)

$est\_thr(a_n, c_n, j_n) = thr(a_n, c_n, l_n) \times (powl(j_n) / powl(l_n))^{1/3}$

B. (*Sum up the total throughput and power for this combination of power allocation levels*)

$T_{total} = 0$, $P_{total} = 0$.

For k from 1…n

$T_{total} = T_{total} + est\_thr(a_k, c_k, j_k)$.

$P_{total} = P_{total} + powl(j_k)$.

End.

C. (*If the total power of this configuration is below budget and it has higher throughput than the current best, make it the best*)

If $P_{total} \le P_{max}$ and $T_{total} > T_{best\text{-}total}$

I. $T_{best\text{-}total} = T_{total}$.

II. $P_{best\text{-}total} = P_{total}$.

III. (*Set the power level assignment and power allocation to this new found best configuration*)

For k from 1…n

$gpm(c_k) = j_k$.

$alloc\_pow(c_k) = powl(j_k)$.

End.

End.

End.

End.

End.

End.

3. Return (gpm(), alloc_pow(), $T_{best\text{-}total}$, $P_{best\text{-}total}$).

We also implement another scheme, called *even power allocation (EPA)*, which simply enforces a policy of allocating the same power to each core throughout the simulations. This allows us to gain insight into how much benefit is obtained by heterogeneously sharing the power among the cores.

5.2.3. Up/Down DVFS

For our new power allocation schemes, the core-level of the power management hierarchy must be more sophisticated than for MaxBIPS. Each core is assigned a power target at the beginning of the management interval and during the interval the DVFS controller associated with the core must monitor the power dissipation level and adjust the voltage and frequency to stay as close to the target as possible. This allows the core to achieve the highest performance for its power budget. Using on-chip voltage regulators, the frequency of these adjustments can be as fine as

100 MHz, or every 10 ns [64]. In this paper, we explore two rates of DVFS for our scheme. One adjusts the voltage and frequency every 100 μs (which could be performed by either an off-chip or on-chip regulator), while the other has 5 μs DVFS periods (requiring an on-chip regulator). The advantage of our power allocation DVFS is that it is capable of more rapidly adapting to program phases to maximize throughput but does not burden the global manager with a more complex optimization or require shorter global intervals.

The algorithm we use for managing per-core DVFS in our power allocation schemes is called *Up/Down DVFS*. Since it is running very frequently and would likely be implemented in hardware, it must be very simple. For this algorithm, we use 17 voltage levels equally spread out between 0.7 V and 1.0 V. The frequency is set to 17 corresponding levels that depend on the core's nominal frequency. See Figure 5.8 for additional definitions specific to Up/Down DVFS.

UD_VR = {0.7, 0.725, 0.75, 0.775, …, 0.975, 1.0}: The range of voltages for Up/Down DVFS to which the cores can be set. We divide the region of 0.7V to 1.0V into 16 parts yielding 17 possible voltage levels spaced out by 0.025V.

U = {1, 2, …, u, … 17}: The range of voltage levels of a core running Up/Down DVFS. The voltage level can range from 1 to 17 and corresponds to the 17 values in VR.

vmap(u) = v: A mapping from a voltage level, u, in U to a voltage, v, in UD_VR for Up/Down DVFS.

t: The time between DVFS periods where Up/Down DVFS checks the core power and adjusts the voltage level.

Figure 5.8: Additional definitions for Up/Down DVFS.

The large number of levels in Up/Down DVFS does not pose a computational problem because the algorithm does not perform exhaustive search. Instead, the algorithm uses the core's current sensors to monitor the core power dissipation in relation to the target power given by the GPM. If the algorithm detects a violation over the past interval, it decrements the voltage and frequency by one level. After a violation, the algorithm waits four intervals before declaring the violation over and

raising the voltage back one level. If another violation of the target occurs in the interim, the power level is lowered another step and the wait period is reset. On the other hand, if after raising the voltage and frequency, there is no violation for two intervals, the algorithm will raise the voltage and frequency another step. The asymmetric waiting periods for lowering and raising the voltage and frequency ensure that the core spends most of its time below but very close to the target power. See below for the pseudocode for Up/Down DVFS:

Input:

(1) The number of periods, n, for which to run the algorithm.

(2) A core, c, and the application, a, assigned to it.

(3) The power level, l, assigned to the core by the TAPAS algorithm.

(4) The amount of power, P_{target}, allocated to the core by TAPAS.

(5) The initial voltage level, u, for the core.

(6) The function *vmap()* mapping voltage levels in U to voltages in UD_VR.

(7) The time, t, between DVFS periods where Up/Down DVFS checks the core power and adjusts the voltage level.

Output:

(1) The final voltage level, u, which the core is set to at the end of the power management interval.

(2) The final voltage, v, which the core is set to at the end of the power management interval.

Steps:

1. (*Set the initial number of periods waited to zero*) w = 0.
2. (*Run Up/Down DVFS for n periods where the power level of the core is monitored and the voltage is dynamically adjusted to meet the prescribed target*)

For i from 1…n:

A. (*Sample the power dissipation of the core*)

pow(a, c, l) = sam_pow(a, c, l).

B. (*If there is a power overshoot, reduce the core voltage. If the power dissipation is below the power target, check if it is time to increase the voltage*)

If pow(a, c, l) > P_{target},

I. (*Reduce the voltage level of the core*) u = u – 1.

II. (*Correspondingly reduce the voltage*) v = vmap(u).

III. (*Set the number of periods to wait*) w = 4.

Else,

I. (*If the waiting period is over, increase the voltage*)

If w == 0,

a. (*Increase the voltage level of the core*) u = u + 1.

b. (*Correspondingly increase the voltage*) v = vmap(u).

c. (*Set the number of periods to wait.*) w = 2.

(*Otherwise, subtract one from the number of periods to wait*)

Else,

w = w - 1.

End.

End.

C. Wait for time t.

End.

3. Return (u, v).

5.3. Methodology

For this study, we use the same infrastructure as described in Section 4.3, augmented with the capability to run global power management algorithms and DVFS. We add support for variable voltage and frequency levels to our improved version of the SESC simulator [107] as well as implement the Up/Down DVFS algorithm for per-core power control. We assume linear scaling of frequency with voltage when employing DVFS and scale voltage from a nominal value of 1.0V down to 0.7V. The clock frequency ranges from a maximum of 4.0 GHz down to 2.8 GHz for a baseline core. The baseline architectural parameters for the cores are outlined in Table 5.2.

Table 5.2: Core architectural parameters.

Front-End Parameters	
Branch Predictor	Hybrid of gshare and bimodal with 4K entries in the bimodal, gshare 2nd level, and meta predictor
Branch Target Buffer	512 entries, 4-way associative
Return Address Stack	64 entries, fully associative
Front-End Width	4-way
Fetch Queue Size	32 entries
Re-Order Buffer	128 entries
Retire Width	4-way
Back-End Parameters	
Integer Issue Queue	48 entries, 4-way issue
Integer Register File	80 registers
Integer Execution Units	4 ALUs/address calculation units and 1 multiply/divide unit
Floating Point Issue Queue	24 entries, 1-way issue
Floating Point Register File	80 registers
Floating Point Execution Units	1 adder and 1 multiply/divide unit
Memory Hierarchy	
L1 Instruction Cache	8KB, 2-way associative, 1 port, 1 cycle latency
Instruction TLB	32 entry, fully associative, 1 port
Load Queue	48 entries, 4 ports
Store Queue	24 entries, 2 ports
L1 Data Cache	8KB, 2-way associative, 2 ports, 1 cycle latency
Data TLB	32 entry, fully associative, 2 ports
L2 Cache	1MB, 8-way associative, 1 port, 10 cycle hit latency, 5 cycle miss latency
Main memory	1 port, 200 cycle latency

We model unpredictably heterogeneous chip multiprocessors in a similar manner to Chapter 4, with three types of degradations: hard faults that require disabling part of the core, frequency asymmetry, and leakage power variation; the last two effects are due to process variations. In this study, we assume a core's clock speed can be degraded by as much as 30%, and that cores are binned such that they are clocked at 100%, 90%, 80%, or 70% of the nominal frequency. This simplifies the process of testing the chips and setting their clock frequencies. From this set of possible effects, a vast number of heterogeneous chip multiprocessors can be created. We randomly generate three degraded CMPs for this study which are listed in Table 5.3. Each core is affected by any or all of the three types of degradations.

Table 5.3: List of faults and variations affecting each core in the 3 degraded CMPs.

Core	Degraded CMP 1	Degraded CMP 2	Degraded CMP 3
1	3.6 GHz, 2X LSQ leakage	½ size Inst. L1, 3.6 GHz, 2X L1 cache leakage	doubled memory latency, ½ size ROB, 3.2 GHz, 2X floating point leakage
2	½ size load queue, 3.2 GHz	2 broken integer ALUs, 3.6 GHz, 2X L1 cache leakage	½ size Inst. L1, 2X floating point leakage
3	½ size L2 cache, 2.8 GHz, 2X LSQ leakage	½ integer IQ bandwidth, 2X L1 cache leakage	3.2 GHz, 2X core leakage
4	½ size load queue, 3.6 GHz, 2X floating point leakage	2X L1 cache leakage	½ size Inst. L1, 2X LSQ leakage
5	doubled memory latency, 2X L1 cache leakage	doubled memory latency, 2.8 GHz, 2X front-end leakage	½ load queue ports, ½ size load queue, 3.2 GHz, 2X L1 cache leakage
6	doubled memory latency, 3.2 GHz, 2X core leakage	2 broken int. ALUs, 3.6 GHz, 2X integer cluster leakage	½ integer IQ bandwidth, 2 broken integer ALUs, 2.8 GHz, 2X front-end leakage
7	half bandwidth front end, 2X floating point leakage	½ size integer IQ, 3.6 GHz, 2X floating point leakage	½ size L2 cache, 3.6 GHz, 2X cache leakage
8	½ size L2 cache, 2X floating point leakage	½ integer IQ bandwidth, 2.8 GHZ, 2X LSQ leakage	½ size integer IQ, 2.8 GHz, 2X front-end leakage

We tested our power management and scheduling policies on four randomly generated eight-threaded workloads of SPEC CPU2000 applications which were run on each of our degraded CMPs. The workloads we used are shown in Table 5.4. For each simulation, we fast forwarded every benchmark two billion instructions, and then executed one billion cycles in SESC, or 0.25 seconds at a nominal frequency of 4 GHz. Cores that run at lower frequencies execute for proportionally fewer cycles.

Table 5.4: Application workloads.

Workload 1	apsi, crafty, mcf, mesa, mgrid, vortex, vpr, wupwise
Workload 2	ammp, crafty, mcf, mesa, mgrid, swim, vpr, wupwise
Workload 3	ammp, applu, art, bzip2, mgrid, twolf, vortex, wupwise
Workload 4	applu, bzip2, crafty, equake, parser, swim, vortex, wupwise

5.4. Results and Discussion

In this section, we present the performance results for the power management schemes. We ran each power management policy with each of the four application workloads on all three degraded eight core CMPs. All results compare the algorithms based on the total throughput across the eight applications. Every algorithm is very effective at keeping the power within the chip-wide budget, which is set to 60W. The percentage of time in violation is always less than 1% and the majority of the time it is under 0.1%.

We compare the power management algorithm by two different criteria. First, we examine how MaxBIPS and our newly proposed power allocation schemes fare compared to a baseline GPM of Chip-Wide DVFS. Then we investigate how effective the various algorithms are at recovering the performance lost relative to not employing power management (and letting the chip overheat). This criterion provides perspective on the maximum amount of lost performance that could possibly be made up by a better power management policy.

Figure 5.9 shows the throughput improvements of the more advanced power management algorithms compared to Chip-Wide DVFS organized by degraded configuration, with each bar representing the average across all four workloads. Implementing per-core DVFS provides almost 6% improvements across the degraded configurations when employing the MaxBIPS algorithm. In the simple Even Power Allocation algorithm, the GPM just assigns 1/8 of the power budget (7.5W) to each core and then uses our Up/Down DVFS algorithm to keep the power as close to that target as possible. Despite this algorithm's simplicity, it out-performs MaxBIPS by almost 2% and Chip-Wide DVFS by 7.8% on average. The EPA algorithm also has the advantage of requiring neither exhaustive search nor linear optimization, both of which add substantially to the implementation cost and runtime overhead of the GPM.

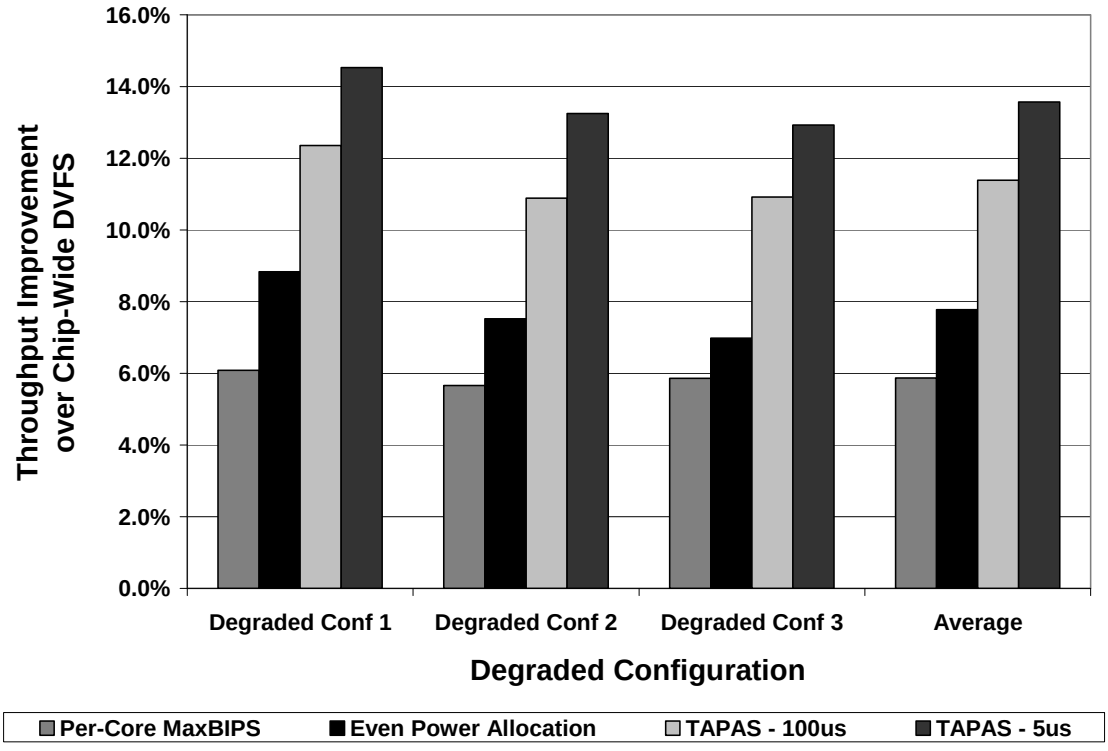


Figure 5.9: Throughput improvement for the global power management schemes across the degraded configurations.

The TAPAS algorithm is even more effective, beating Chip-Wide DVFS by an average of 11.4% and 13.6% when implemented with Up/Down DVFS running with 100 μs and 5 μs intervals, respectively. TAPAS is also 5.2% and 7.3% better than MaxBIPS with 100 μs and 5 μs Up/Down DVFS intervals. This shows that exploiting the potential of fine-grained DVFS has advantages, but that this feature of on-chip voltage regulators is not essential to effectively utilize TAPAS.

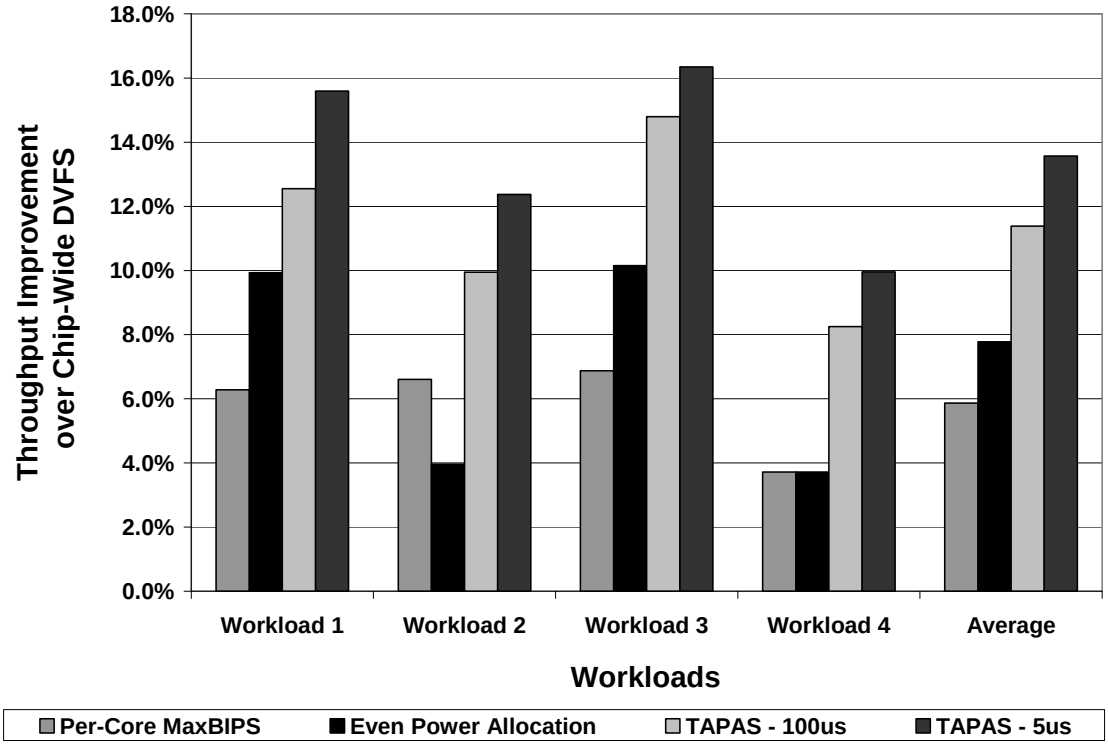


Figure 5.10: Throughput improvement for the global power management schemes across the benchmark workloads.

One interesting aspect is how consistent the results are across degraded configurations. Examining Figure 5.10, we see that the relative performance of the GPM algorithms changes much more significantly across the four workloads than across the configurations. In particular, the Even Power Allocation scheme is more suitable to some workloads than others. This occurs because some benchmarks in

these workloads do not make effective use of the 7.5W, wasting power that could be allocated to benchmarks where it would result in a bigger throughput improvement.

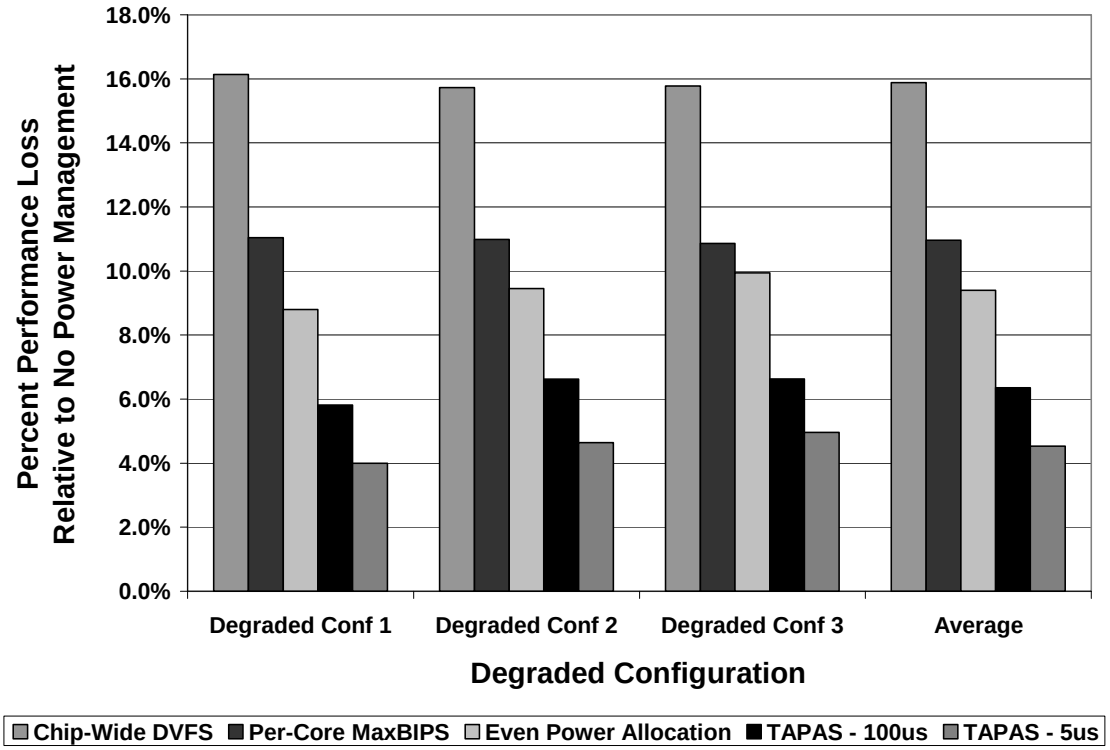


Figure 5.11: Percent performance lost relative to no GPM across the degraded configurations.

Next, we compare our algorithms in term of their capacity to recover the performance lost by the need to abide by a chip-wide budget. Figure 5.11 shows that implementing the simple Chip-Wide DVFS policy that does not require on-chip voltage regulators reduces throughput by almost 16% compared to employing no power management. However, implementing the 5 μs interval variant of TAPAS reduces that performance loss to only 4.5%, recovering over 70% of the lost throughput overhead of meeting the power budget. Figure 5.12 shows that while TAPAS is consistently the most effective GPM scheme, MaxBIPS can be better or worse than Even Power Allocation, depending on the level of heterogeneity in the power demands of the application workload.

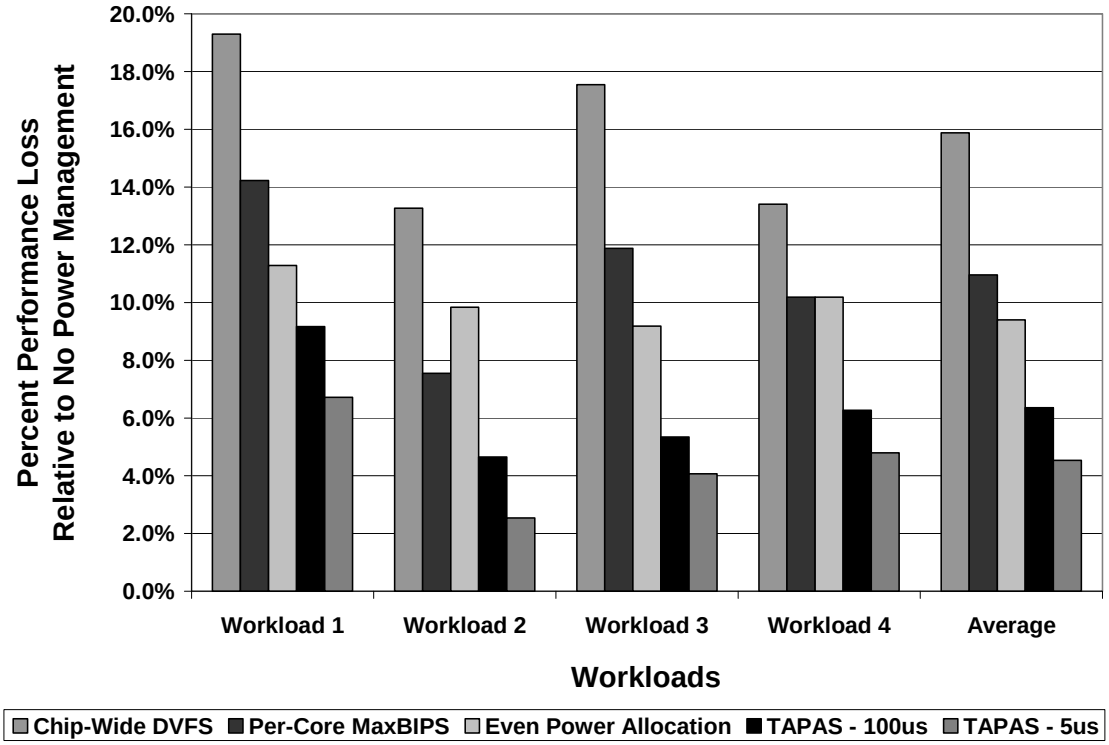


Figure 5.12: Percent performance lost relative to no GPM across the benchmark workloads.

5.5. Possible Extensions

This chapter makes a clear case for the merits of power allocation techniques for global power management and for TAPAS in particular. However, there are a number of avenues for future work. First, a more comprehensive comparison to prior global power management work would help make a stronger case for the newly proposed approach. Key algorithms to compare against would include LinOpt [134], the Greedy and Threshold schemes of Herbert and Marculescu [53], the Model Predictive Control theory approach [137], and Steepest Descent [88]. One aspect to investigate would be to compare the GPM algorithms in terms of their ability to meet the global power target and their amount of time in violation of this budget. The actual success of the algorithms at meeting the target is obviously very critical but mostly overlooked in prior work. Other metrics for evaluating TAPAS include runtime

overhead and scalability. Future work should extend Chapter 7's scalability study to include power allocation schemes and compare TAPAS and Even Power Allocation against traditional approaches that set DVFS values at the global level.

More study is also possible to improve the TAPAS algorithm. As mentioned above, to fully exploit TAPAS, the allocation mechanism could be allowed to assign continuous values for power to each core rather than the seven discrete levels above. This could improve the performance of TAPAS significantly in comparison to MaxBIPS. Furthermore, although Up/Down DVFS is simple and effective, other implementations of the core-level power manager could be employed (perhaps a version based on formal control theory). Lastly, TAPAS could be explored from an energy efficiency perspective as its ability to track power targets more effectively may translate into ED^2 improvements.

5.6. Conclusions

In this chapter, we develop a very effective global power management technique called TAPAS, based on a new way of thinking about DVFS. Rather than try to guess the power that an application/core pair will consume when assigned a voltage and frequency, our scheme operates by allocating power directly. Voltage and frequency scaling is then managed by each core individually using a simple and effective Up/Down DVFS technique. On unpredictably heterogeneous CMPs, our best TAPAS implementation, achieves 7.3% improvement over the well known MaxBIPS algorithm and a 13.6% throughput increase over Chip-Wide DVFS. This amounts to recovering over 70% of the performance lost by the need to meet a global power budget. In future work, we plan to show that TAPAS is also far more effective at meeting the chip-wide power target as well as more scalable to future many-core architectures.

CHAPTER 6

A SCALABILITY ANALYSIS OF SCHEDULING ALGORITHMS FOR UNPREDICTABLY HETEROGENEOUS CMP ARCHITECTURES

6.1. Introduction

In Chapter 4, we developed operating system scheduling algorithms for unpredictably heterogeneous chip multiprocessors. Our evaluation consisted of running the algorithms with four workloads on a single degraded eight-core CMP configuration. In this section, we broaden our evaluation of our scheduling techniques by exploring more workloads and degraded processor configurations, as well as experimenting with chips containing four to sixty-four cores. Besides validating our results on more test cases, the chief goal of this section is to assess whether the algorithms we designed in Chapter 4 will continue to provide power-performance efficient scheduling for future many-core processors.

An *n* core CMP running *n* applications has n! possible scheduling assignments leading to a rapidly growing search space as the number of cores increases. Scalable algorithms must continue to find effective scheduling assignments despite this dramatic increase in the solution space. Furthermore, these algorithms must function well with a reasonable number of exploration phase sampling intervals and low computational overhead. As mentioned in Chapter 4, the Hungarian Scheduling Algorithm could potentially have serious scalability concerns. First of all, it requires each application to be sampled on each core in the CMP, requiring *n* sampling intervals for an *n* core processor. For large-scale multi-core processors, this sampling time becomes prohibitive, yet there is no way around it for our current implementation of the algorithm. Secondly, the Hungarian Algorithm also has $O(n^3)$ computational complexity. Our iterative optimization search algorithms have lower runtime complexity but their effectiveness is highly dependent on the number of search

iterations they are allowed to conduct. In this chapter, we focus on evaluating the energy efficiency of the Hungarian and iterative optimization algorithms. We also characterize the impact that the number of search iterations has on the iterative algorithms. An in-depth study of the computational complexity of these scheduling algorithms is conducted in the next chapter.

In the next section, we discuss some simplifications to the experimental methodology of Chapter 4 that we made in order to facilitate the large scalability study that we perform. In Section 6.3 we analyze the power-performance efficiency of the various scheduling algorithms, and then evaluate the impact of the length of the exploration phase on the iterative optimization search algorithms. Section 6.4 then concludes this chapter.

6.2. Methodology

The simulation framework for this study is very similar to that used in the previous two chapters. We use the same hierarchical infrastructure to model our multi-core processors, although we extended its capabilities to simulate chip multiprocessors of arbitrary size. We also augmented the system with the ability to generate random workloads and degraded configurations. In this chapter, we use the same baseline non-degraded cores as before and a detailed list of architectural parameters can be found in Table 4.2.

In this study, we evaluate a fixed 40 million cycle window of the execution of each application. This window is used for each round robin rotation interval, Hungarian Scheduling Algorithm sample, search algorithm exploration interval, and the steady-state phases, by not advancing the starting point of the applications for a subsequent simulation. This is the major simplification we made to our infrastructure in comparison to the previous chapter's study. The reason for this simplification is that it allowed us to reuse the single-core SESC simulation results from running a

particular application on a certain degraded core configuration if that combination ever came up again during an exploration interval or steady phase. Since we ran a huge number of workloads, processor configurations, and CMP sizes, many combinations showed up repeatedly. By reusing application/degraded core pair simulations results, we save a tremendous amount of computation, thus allowing a study of this magnitude to be tractable.

In the study in this chapter, the exploration phases of the scheduling policies consist of using the algorithms to decide which application to sample on each core during each interval and then executing a 40M cycle simulation for each interval. After the specified number of exploration intervals, the algorithms are evaluated by calculating the average energy-delay-squared (ED^2) product of the best schedule found by the algorithms over the same 40M cycle interval. One exception is the Round Robin Algorithm, where the mean performance and power values across the *n* rotation steps for an *n* core CMP are used to compute the ED^2 in order to account for the averaging effect of the rotation. The average ED^2 is compared against the average ED^2 of a baseline architecture with homogeneous cores that do not suffer from errors or variations, as well as an oracle scheduler which uses the optimal assignment of applications to cores over the 40M cycle interval.

We examine chip multiprocessors with four, eight, 16, 32, and 64 cores. For each CMP size, we randomly generate four appropriately sized workloads from among 17 SPEC CPU 2000 benchmarks that work on our SESC simulator. These benchmarks include *mgrid* plus those listed in Table 4.4. We consider the same three types of degradations as in Chapter 4, but we add a few extra alternatives for the effects of hard errors as well as a 30% frequency degradation consistent with prior work [13][134]. Table 6.1 presents a list of the core degradations possible for each type of fault. To make the study more feasible, we reduce the space of possible degraded configurations

by assuming a core can be affected by only one problem (or none) from each category of degradation. We then create four unpredictably heterogeneous CMP configurations for each size processor by randomly selecting the appropriate number cores from among the possible degradations. In all, there are $12 \times 4 \times 7 = 336$ such degraded core configurations. With 17 applications and our use of the fixed 40M cycle intervals, we run a total of 5,712 simulations that are reused across the scalability study.

Table 6.1: List of possible forms of core degradation.

Degraded Unit	Frequency Degradation	Leakage Increase
none	none (4 GHz)	none
memory latency is doubled	10% (3.6 GHz)	2X nominal in L1 caches and TLBs
half the L2 cache	20% (3.2 GHz)	2X nominal in front end and ROB
half the L1 Icache	30% (2.8 GHz)	2X nominal in integer back end
front end is reduced from 3-way to 2-way		2X nominal in floating point back end
half the integer issue queue		2X nominal in load and store queues
integer issue bandwidth is reduced to one		2X nominal across core (excluding L2)
one integer ALU is disabled		
half the load queue		
half the store queue		
half the L1 Dcache		
half the re-order buffer		

6.3. Results and Discussion

6.3.1. Scheduling Algorithm Performance

Our goal in this section is to characterize the potential of our scheduling algorithms and gauge how this potential is affected by scaling the number of cores. Consequently, we initially assume that our algorithms have sufficient time to sample different points in the search space. For instance, the Hungarian Scheduling Algorithm is able to obtain all n^2 samples of the execution of each of the *n* applications on each of the *n* cores. For large-scale CMPs, there may not be enough time to obtain all the

samples, unless the samples are very short, but then they may not reflect the long term behavior of the application.

In addition, we report the ED^2 results for the steady phase of the algorithms, ignoring the overhead of sub-optimally executing applications during the exploration phase. An analysis of the execution overhead will be presented in Chapter 7. Figure 6.1 shows the overall results of running each of the scheduling algorithms as the number of cores in the CMP is varied from four to sixty-four. Each bar represents the average increase in ED^2 across 16 different simulations (four workloads run on each of four degraded CMP configurations) relative to a baseline CMP of the same size unaffected by hard errors or variations.

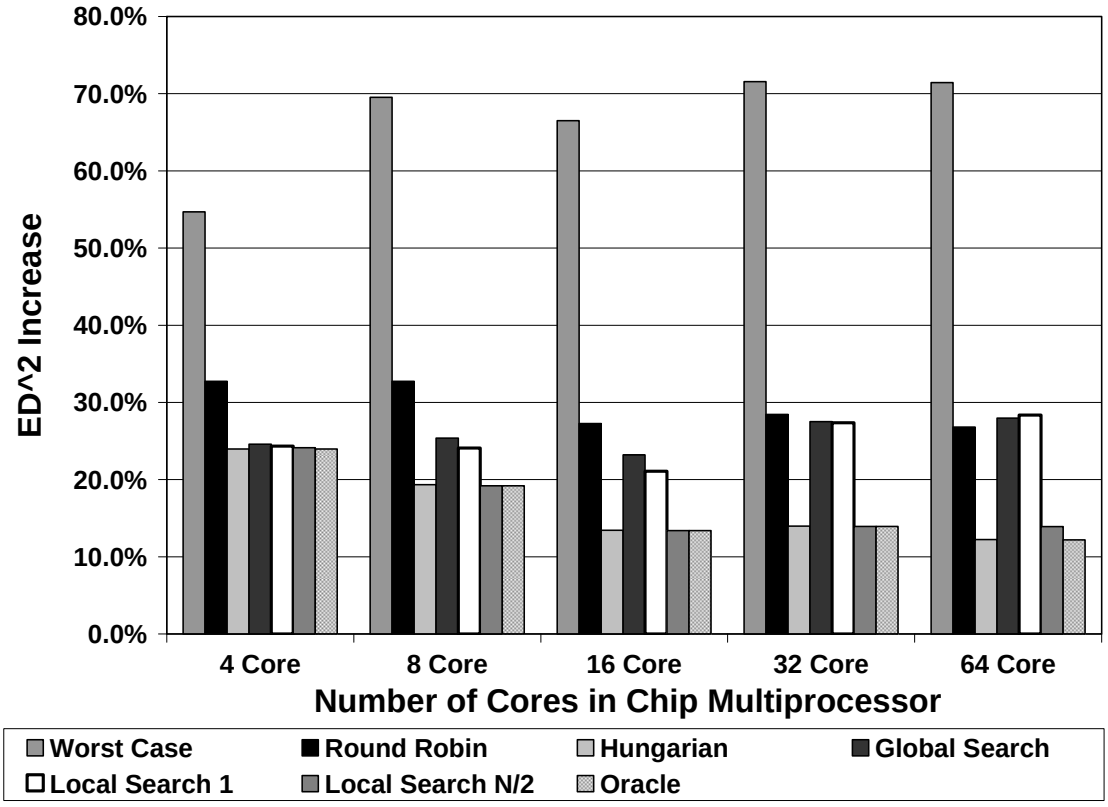


Figure 6.1: The power-performance efficiency results for the scheduling algorithm scalability study.

The first bar in the graph represents the worst case assignment of applications to cores, or the maximum possible loss in power-performance efficiency if programs were assigned to cores randomly without any thought towards core heterogeneity and the affinity of applications for some cores. For the four core designs, the worst case can be as bad as 55% above the baseline with no degradation, reaching 71% worse ED^2 for larger CMPs. The last bar in the graph represents an oracle scheduler that has complete knowledge, including the performance of the applications on the baseline core, and thus can find the scheduling assignment closest in ED^2 to the baseline.

From the results, we see that the naïve Round Robin Algorithm performs quite poorly although much better than the worst case. This algorithm does quite well on homogeneous CMPs and on multi-core architectures which are designed intentionally to be heterogeneous [9]. However, in our scenario, Round Robin's averaging effect is not capable of reclaiming the full potential of the baseline design, leading to energy-delay-squared increase of over 25% across all configurations.

The Hungarian Scheduling Algorithm finds very close to the optimal schedule, coming within 0.15% of the ED^2 of the oracle scheduler. The algorithm also reduces the impact of the core degradation with more cores. It fares best with 64 cores, reducing the ED^2 increase from 26.8% for Round Robin to only 12.3%, reclaiming more than half of the lost efficiency.

For the iterative optimization algorithms, we allow each algorithm to use 25 exploration intervals as in Chapter 4. (Section 6.3.2 shows how varying the number of search intervals affects the success of the algorithm.) Figure 6.1 shows that while Global Search is competitive for smaller CMP configurations, it scales quite poorly, eventually lagging behind the Hungarian Scheduling Algorithm by 14.3% for 64 cores. As the search space increases with more cores, Global Search cannot perform as well when evaluating only 25 randomly sampled schedules.

Local Search 1, which performs one swap to find a neighbor in each exploration interval, initially fares better than Global Search but still fails to scale to larger CMPs. As the number of cores increases, the search space explodes and for 64 cores, it's better to run Global Search and evaluate 25 random configurations than to run Local Search in one corner of the search space.

Local Search N/2 swaps the core assignment of every application with another one in each search interval. With the ability to incorporate good swaps into the best solution and discard poor swaps, Local Search N/2 actually evaluates $2^{N/2}$ scheduling assignments each interval. This means that the algorithm evaluates more assignments as the number of cores grows in the same period of time. This feature may provide Local Search N/2 with the ability to scale to larger multi-core architectures. In our study, this search algorithm's ED^2 increase is never more than 1.7% above the oracle scheduler across all core sizes.

Another insight from this study is that as the number of cores increases, the scheduling algorithms are better able to mitigate performance losses and power dissipation increases from unreliable cores. While the oracle algorithm delivers an ED^2 24% higher than the baseline for a four core chip, it gets progressively better until it is only 12.2% worse for 64 cores. Likewise the oracle achieves an ED^2 8.8% less than Round Robin for four cores, but this improves to 14.6% for 64 cores. One of the reasons that the oracle, Hungarian, and Local Search N/2 schedulers perform better relative to the baseline as the number of cores scales up is that the application and core configuration diversity increases with larger CMPs. With more programs and more degraded configurations, there is more of an opportunity to find a good match between each application and core and reduce the performance and power penalty caused by the errors and variations.

6.3.2. The Impact of the Number of Intervals on the Search Algorithms

This section seeks to evaluate how the number of intervals allocated to the search algorithms affects their performance. In Section 6.3.1, we allocated 25 intervals to each of the three iterative algorithms. In Figure 6.2, we show how these three algorithms perform as the number of intervals is varied from five to 100. The base segment of each bar in the graph shows the power-performance efficiency of the oracle scheduler. This represents the minimal ED^2 increase possible if the search algorithms were given unlimited search intervals to find the best scheduling assignment possible. The next segment shows how well the search algorithms do when given the maximum of 100 intervals to try different points in the search space. Each subsequent bar on top represents the added increase in ED^2 when the algorithms have fewer search intervals. For each of the search segments, the bar represents the results of 160 runs, corresponding to ten runs with different random seeds for each of the four application workloads and four degraded core configurations.

Global Search performs much worse than the oracle for CMP sizes greater than four cores even with 100 search intervals. The search space is simply too big to explore and more random samples are needed to guarantee good results.

When given 100 intervals, Local Search 1 does better than Global Search up to sixteen cores. However, as the search space rapidly increases, the performance of Local Search 1 degrades. Furthermore, reducing the number of sampling intervals significantly increases its ED^2. With only five search intervals, Local Search 1 does even worse than Global Search because it explores a tiny segment of the space.

Local Search N/2 holds the most promise for scaling to larger CMPs. With 100 search intervals, its performance is never more than 0.6% above the oracle. Local Search N/2 continues to do well even with only 50 and 25 intervals, but the ED^2 starts to increase significantly when only ten and five search intervals are permitted. Local

Search N/2 has two potential advantages over the Hungarian Scheduling Algorithm. First, it only needs to perform O(n) computation for each search interval (compared to Hungarian's $O(n^3)$) to generate random swap pairs and evaluate if the swapped applications performed better or worse on their new core assignment. Second, by evaluating $2^{N/2}$ scheduling assignments in parallel every search interval, Local Search N/2 may be able to achieve good results with a shorter exploration phase than the Hungarian Scheduler's requisite N samples.

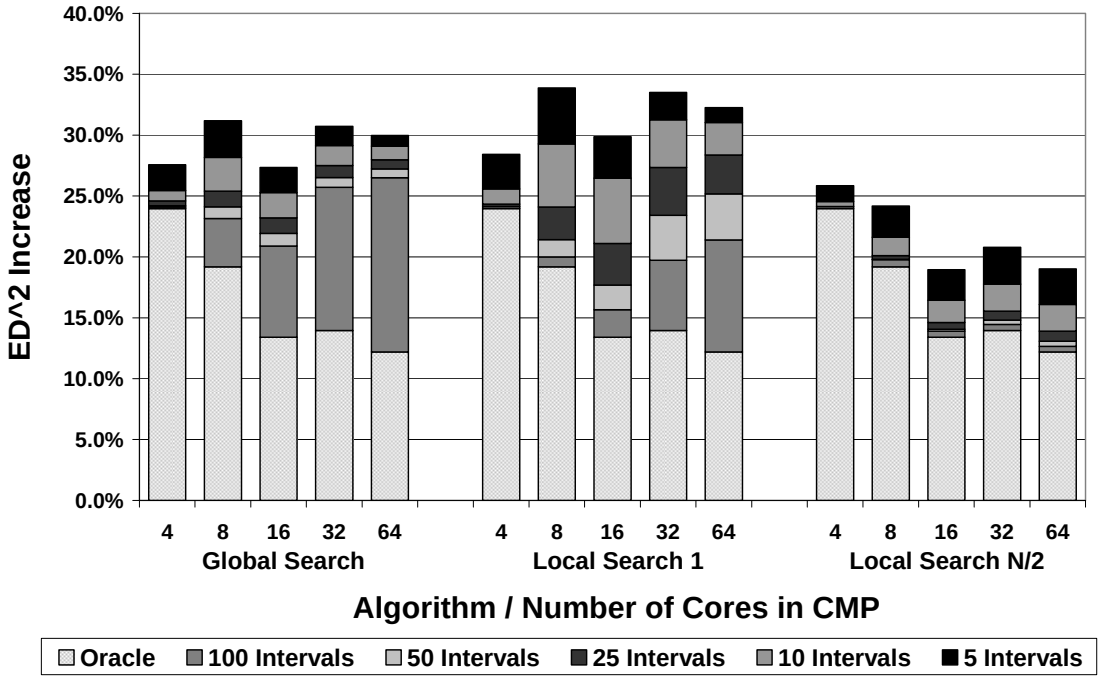


Figure 6.2: The impact of the number of intervals on the effectiveness of the search algorithms.

6.4. Conclusions

In this chapter, we explore the scalability of the scheduling algorithms for unpredictably heterogeneous CMPs that we proposed in Chapter 4. We find that Global Search and Local Search 1 fail to scale effectively because they do not evaluate a sufficiently large portion of the search space during their exploration phases. Consequently, we do not study these algorithms further in Chapter 7. The Hungarian

Scheduling Algorithm achieves energy-delay-squared values extremely close to the oracle scheduler but suffers from the need to collect a large number of samples during the exploration phase and its $O(n^3)$ runtime. Local Search N/2 also holds promise for scheduling in large-scale chip multiprocessors but further analysis of its exploration phase sampling requirements and computational cost are warranted.

CHAPTER 7

SCALABLE THREAD SCHEDULING AND GLOBAL POWER MANAGEMENT FOR FUTURE UNRELIABLE MANY-CORE ARCHITECTURES

7.1. Introduction

This chapter serves as a capstone of our work on adaptive thread management for unpredictably heterogeneous chip multiprocessors. While Chapter 4 looks at application scheduling algorithms and Chapter 5 examines global power management, this chapter studies the need for coordination between these two system managers. Additionally, extending the study of Chapter 6, which identified the Hungarian Scheduling Algorithm and Local Search N/2 as the most scalable scheduling algorithms, this chapter presents a comprehensive scalability study of both thread scheduling and global power management. The offline methodology of Chapter 6 is leveraged to allow for a wide ranging investigation of many-core processors with up to 256 cores. We also validate this offline study with a set of online simulations which confirm the observed performance and scalability trends.

A key aspect of both scheduling and power management, mostly ignored in prior work, is the amount of overhead incurred by the algorithm. The algorithm gathers information in hardware over some period of execution and makes decisions at the end of each period. Many recent approaches use a period of tens to hundreds of milliseconds so that decisions can be made by the operating system (e.g., [53][58][119][134]). For small-scale multi-core microprocessors operating at this interval granularity, the decision overhead may be quite reasonable, even for brute-force algorithms that use exhaustive search (e.g., [58]). However, the move to many-core architectures brings the scalability of these prior algorithms into question, and calls for a detailed investigation into scalable scheduling and power management

algorithms that account for both deconfiguration due to hard errors and differing per-core frequency and leakage characteristics due to variability.

This chapter addresses scalable scheduling and power management algorithms – and their coordination – for future dynamically heterogeneous many-core microprocessors with hundreds of processor cores. In particular, this chapter makes the following contributions:

- A wide-ranging study of the performance, power, and scalability of a variety of potential scheduling and power management algorithms for unpredictably heterogeneous many-core architectures with up to 256 cores that are degraded due to both hard errors and variability;
- A detailed description of each algorithm in pseudocode and a formal complexity theory analysis of the runtime of the algorithms;
- An experimental assessment of the need for coordination between scheduling and power management;
- The identification of highly scalable scheduling and power management algorithms for dynamically heterogeneous many-core architectures that achieve close to optimal performance given a maximum chip power constraint.

The rest of this chapter is organized as follows. In the next section, we provide an overview of the problem of adaptive thread management of future unreliable many-core architectures. Section 7.3 discusses a range of potential scheduling and power management algorithms based on formal techniques, and proposes new, more scalable, approaches suitable for many-core systems. In Section 7.4, we describe our evaluation methodology. Our results section (Section 7.5) first assesses the need for coordination among scheduling and power management, and then evaluates the performance, power, and scalability of these algorithms. Finally, Section 7.6 concludes the chapter.

7.2. Adaptive Thread Management for Unreliable Many-Core Architectures

Power dissipation poses a major challenge for future many-core architectures. As the number of cores grows to hundreds on a single die, ensuring power-

performance efficiency creates a complex optimization problem for the runtime manager. This challenge will only be further exacerbated by the dynamic heterogeneity created by manufacturing faults, wear-out related errors, and process variations. Two key adaptive thread managers, the application scheduler and the global power manager, are chiefly responsible for controlling the operation of the applications running on the cores. Both the scheduler and power manager operate over a quantum of time which consists of two phases, a short *sampling* period and a longer *steady-state* period. During the sampling period, the performance and power statistics of the applications and heterogeneous cores are assessed by running different scheduling assignments (for the scheduler) or power settings (for the power manager) over smaller *intervals* of time. The manager then employs an algorithm to use these interval statistics to make a decision – a scheduling assignment or DVFS settings – at the end of the sampling period. This decision is maintained for the steady-state period until the next quantum. Figure 7.1 describes this process, assuming a 100ms quantum for application scheduling and a 10ms quantum for power management, in line with prior work [53][58][119][134]. We employ sampling periods lasting 10% of the quanta, leaving the other 90% for the steady-state period. This gives the scheduler and power manager 10ms and 1ms, respectively, for their sampling periods.

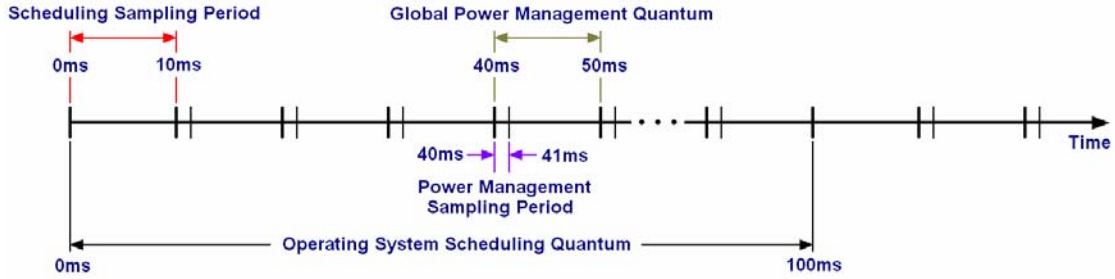


Figure 7.1: Scheduling and power management time quanta and sampling periods.

Regarding the implementation of the application scheduler and global power manager, there is no consensus in the research community about whether these

operations should be performance in hardware, software, or a combination of the two [58][62][114][131][134]. While the operating system is traditionally responsible for application scheduling, the need to sample application behavior on cores at finer granularities than OS context switch times may necessitate hardware support. Some authors [131][134] argue that a dedicated hardware unit should be responsible for the global level of the power manager, and cite the Foxton controller in Intel's Itanium II [84] as an example of how this can be successfully implemented. We suggest that the application scheduler may be implemented at a lower level than the operating system, such as a virtualization layer, a low level hypervisor, or microcontroller in order to hide the burden of addressing the affects of hard errors and variations from system software. Two recent studies [62][114] have argued for distributing the task of power management among the cores to reduce the complexity of the problem. We explore the concept of distributed management for power management and also apply it to thread scheduling. However, it must be recognized that in order to meet a chip-wide power budget, there must be some level of global coordination among the distributed managers in order to communicate performance and power information across the processor.

7.2.1. Scalability Issues for Adaptive Thread Management Algorithms

For the application scheduler and global power manager to operate effectively, the performance and power statistics taken during the sampling period must be reflective of the true application behavior on the processor cores. This requires the manager to have sufficient time to take enough samples, each of reasonable length, to prevent thread migration effects, thermal time constants, and other effects of moving applications and changing power settings from dominating the statistics. Furthermore, the runtime of the algorithm used to make the decision must be short relative to the quantum. Otherwise, the steady-state period will be consumed by the algorithm's

execution and little time will be left to run in the selected scheduling assignment or designated power settings. In this chapter, we investigate how these dual issues of sufficient sampling time and algorithm runtime are impacted by scaling to hundreds of cores on a chip. One might consider increasing the length of the quanta as the many-core processor increases in size. However, larger quanta will make it harder to adapt to the changing phase behavior of the running applications, a problem that will only increase as the processor executes more and more applications simultaneously. A study of the impact of length of the scheduling and power management quanta is left for future work.

Regarding algorithm execution time, a fundamental method for assessing algorithm scalability is to derive its computational complexity. In this work, we analyze the computational complexity of each scheduling and power management algorithm and then provide experimental results corroborating these findings. Traditionally, polynomial time algorithms are considered sufficiently efficient for scalability. However, in our scenario where the adaptive thread manager is tasked with making decisions tens or hundreds of times per second for architectures with hundreds of cores, we show that even $O(n^3)$ and $O(n^4)$ algorithms do not successfully scale. In order to provide intuition for the importance of algorithm complexity, Figure 7.2 shows a comparison of the growth in runtime of algorithms of different complexity as the number of cores on the chip is increased. In this abstraction, we assume there are no constant factors and that the unit for measuring complexity is the number of processor cycles required to compute the solution to the scheduling or power management problem. It can be seen from this graph that algorithms with factorial ($O(n!)$) or exponential ($O(7^n)$) complexity rapidly become extremely time-consuming to run even for a processor with sixteen cores, making them poor candidates for future many-core architectures. Likewise, an $O(n^4)$ algorithm would take over one billion

cycles (250ms on a 4GHz processor) at 256 cores, making it impossible to be employed at millisecond granularities. Even an algorithm with $O(n^3)$ complexity will require tens of millions of cycles to execute, bringing into question its feasibility.

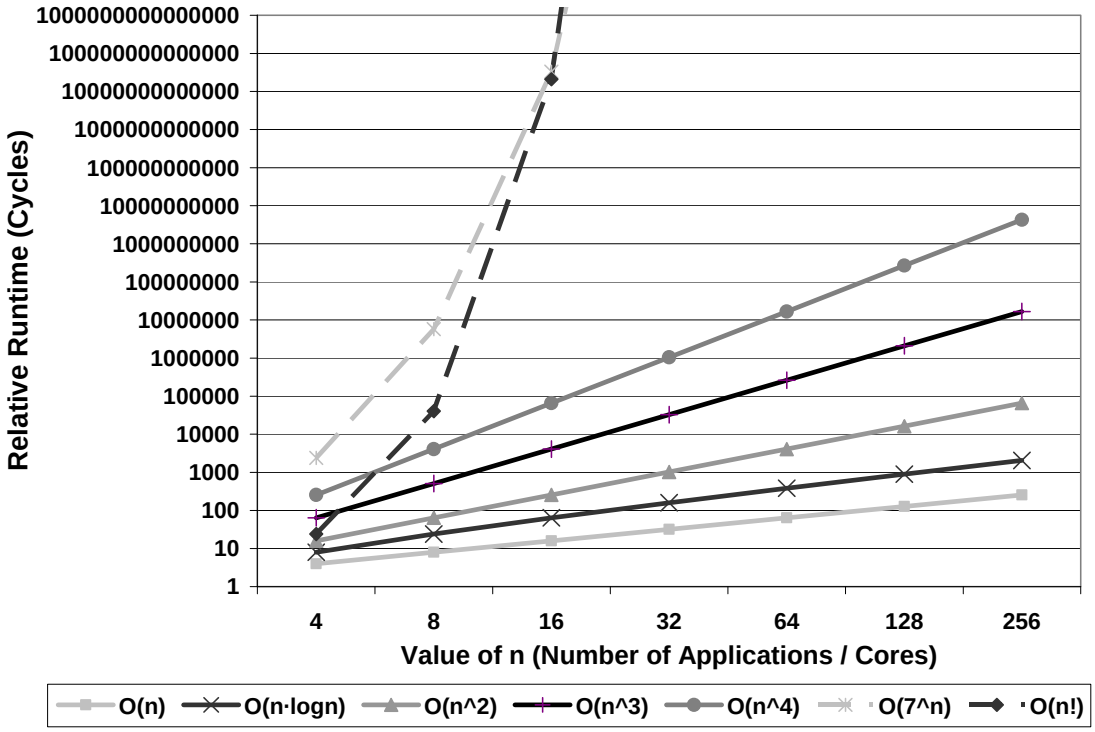


Figure 7.2: The growth in the runtime of various algorithm complexity classes.

7.2.2. Coordinating Application Scheduling and Global Power Management

Future unpredictably heterogeneous many-core processors will require both intelligent scheduling and intelligent power management algorithms that are aware of hardware degradations in order to mitigate their performance loss. A key question in terms of scalability, thus far unaddressed in the research community, is whether a lack of *coordination* between the two algorithms significantly degrades performance or whether the two algorithms produce good results working independently. One might presume that since both scheduling and power management adjust the performance and power profile of the running applications, there could be destructive interference

between their decisions. On the other hand, if no coordination is necessary, then the overhead of runtime management is greatly reduced because scheduling and power management can be optimized separately, thereby avoiding exploring the combined search space of both problems.

We propose that scheduling and power management can in fact be performed independently with little loss in efficiency and validate this claim in Section 7.5.1. The key intuition regarding the lack of interference comes from understanding how scheduling and power management affect application performance. In unpredictably heterogeneous many-core architectures, the runtime of thread *i* on core *j* can be considered a function of four components described in the following equation:

$$Runtime(i,j) = IPC(i,j) \times Base\_Freq \times Var\_Freq\_Scale(j) \times DVFS\_Freq\_Scale(j)$$

The first component is instructions per cycle (IPC) which is a function of the instruction-level parallelism (ILP) available in the application and its memory access patterns, as well as the degree to which hard errors in the core affect the execution of the application on that core. The base frequency of each core is assumed to be the same in our many-core processor since we start with an architecture that was designed as a homogeneous system. The third component is a scaling factor results from the impact of process variations on the frequency due to reduced transistor switching speeds. Together these three components dictate the *inherent performance capability* of an application on a core. The fourth component is a factor taking into account dynamic voltage and frequency scaling, which allows the core to operate at a range of frequencies below the core's maximum inherent frequency established by the architecture and impacted by variability. While changing frequency has some impact on IPC due to off-chip memory access and other asynchronous activity, for the most

part, DVFS affects application runtime by altering core frequency rather than influencing IPC.

If the DVFS levels for all the cores on the chip were held constant, the application scheduler would optimize for the inherent performance capability of the applications on the cores. The resulting performance values would be modulated by the power management algorithm seeking to meet a power target by adjusting voltages and frequencies without impacting the benefit of the scheduling assignment. Thus, scheduling and power management can be considered to tackle different elements of the application/core performance equation. In order to fairly assess different scheduling options, our application schedulers always sample applications at *isometric* voltage and frequency levels to make DVFS independent decisions. We set all cores to the middle DVFS level to avoid exceeding the power budget during the scheduler's sampling period.

7.3. Application Scheduling and Global Power Management Algorithms

7.3.1. Overview

The tasks of determining the best assignment of applications to cores and determining the optimal voltage/frequency settings in a many-core processor are essentially large-scale optimization problems. The optimization challenges can be approached from a number of perspectives. In this chapter, we make an effort to be comprehensive and present a variety of algorithms for both problems that cover the basic styles of optimization. First, we discuss brute force approaches that find the optimal solution but have major scalability limitations. Next, we examine greedy approaches designed to be simple and fast to provide high levels of scalability. We then develop heuristic techniques based on well-known methods in combinatorial optimization. Next, we study linear programming, a classical and effective approach

for solving a wide range of optimization problems. Finally, we consider hierarchical algorithms designed to cut down the complexity of managing many-core processors. Both the sampling requirements and computation complexity of the algorithms are assessed, as both components must scale efficiently to ensure the algorithms' feasibility in future many-core architectures. Figure 7.3 presents the formulas used in the algorithms that are based on the assumptions from Figure 5.2. Figure 7.4 presents a list of definitions of terms used throughput the algorithms described in this chapter.

$$thr(a_i, c_i, l_i) = ipc(a_i, c_i, l_i) \times freq(c_i, l_i) \quad (1)$$
$$est\_thr(a_i, c_i, l_i) = thr(a_i, c_i, m) \times volt(l_i) / volt(m) \quad (2)$$
$$est\_pow(a_i, c_i, l_i) = pow(a_i, c_i, m) \times volt(l_i)^r / volt(m)^r \quad (3)$$
$$pp\_slope(a_i, c_i, l_j) = (pow(a_i, c_i, l_i) - pow(a_i, c_i, l_i\text{-}1)) / (thr(a_i, c_i, l_i) - thr(a_i, c_i, l_i\text{-}1)) \quad (4)$$

Figure 7.3: Formulas used in the adaptive thread management algorithms.

7.3.2. Application Scheduling Algorithms

Unpredictably heterogeneous many-core processors present a distinctly challenging scheduling problem. As shown in Chapter 4, this asymmetric core scenario is highly amenable to sophisticated scheduling policies that greatly improve performance-power efficiency over contemporary OS schedulers and those developed for architected heterogeneity. A major reason for the different needs of unpredictable and architected heterogeneity is that asymmetry resulting from variations, manufacturing defects, and wear-out cannot be anticipated at runtime and manifests itself in myriad ways. The number of possible distinctly degraded cores increases exponentially with the number of failure modes. Consequently, scheduling algorithms must be robust and broadly applicable.

.

n: The number of applications as well as cores in the system.
$A = \{a_1 \ldots a_n\}$: The set of applications in the system.
$C = \{c_1 \ldots c_n\}$: The set of cores in the system.
$sch(a_i) = c_i$: A function representing the scheduling assignment mapping application, a_i, to core, c_j. This function is one to one and onto.
p: The number of power levels in the system.
$L = \{1 \ldots p\}$: The set of power levels in the system. In our case p = 7.
m: The middle power level, $m = \lceil p / 2 \rceil$. In our case m = 4.
$gpm(c_i) = l_j$: A function representing the global power management assignment mapping core c_i to power level l_j.
$VR = \{0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0\}$: The range of voltage levels to which the cores can be set. These levels correspond to the power levels in L.
$volt(l_i) = v_i$: A function mapping power level, l_i, to voltage setting, v_i. In our case, volt(1) = 0.7V, volt(2) = 0.75V, volt(3) = 0.8V, volt(4) = 0.85V, volt(5) = 0.9V, volt(6) = 0.95V, and volt(7) = 1.0V.
FR: The range of frequency levels to which the cores can be set**.** These levels are real-valued and correspond to the power levels in L.
$freq(c_i, l_j) = f_k$: A function mapping core, c_i, and power level, l_j, to frequency, f_k. Due to variations, the frequency that a power level maps to depends on the core (unlike the voltage). Frequencies are real-valued and measured in Hertz.
P_{max}: The chip-wide power budget.
$ipc(a_i, c_i, l_j) = i_k$: A function representing the instructions per cycle (IPC), i_k, of application, a_i, on core, c_i, at the power level, l_j. IPC, i_k, is real-valued and is unitless.
$sam\_ipc(a_i, c_i, l_j) = i_k$: This function represents taking a sample of the instructions per cycle (IPC) committed of application, a_i, on core, c_i, at the power level l_j running in the chip multiprocessor.
$thr(a_i, c_i, l_j) = t_k$: A function representing the calculated throughput, t_k, of application a_i on core, c_i, at the power level, l_j. Throughput, t_k, is real-valued and is measured in the units billions of instructions per second (BIPS).
$est\_thr(a_i, c_i, l_j) = t_k$: A function representing the estimated throughput, t_k, of application, a_i, on core, c_i, at the power level l_j. Throughput, t_k, is real-valued and is measured in the units billions of instructions per second (BIPS).
$pow(a_i, c_i, l_j) = p_k$: A function representing the power, p_k, of application, a_i, on core, c_i, at the power level l_j. Power, p_k, is real-valued and is measured in watts.
$sam\_pow(a_i, c_i, l_j) = p_k$: This function represents taking a sample of the power of application a_i on core c_i at the power level l_j running in the chip multiprocessor. Power, p_k, is real-valued and is measured in watts.
$est\_pow(a_i, c_i, l_j) = p_k$: A function representing the estimated power, p_k, of application, a_i, on core, c_i, at the power level l_j. Power, p_k, is real-valued and is measured in watts.
r: This is the exponent in the model for relating power to voltage. In the standard model, r = 3, but a more aggressive implementation uses r = 2.5.
rand(S): Returns a random element of the set S.
$pp\_slope(a_i, c_i, l_j) = s_k$: The power-performance slope of application, a_i, on core, c_i, at the power level, l_j. Slope, s_k, is real-valued.
heap: A max-heap data structure for storing power-performance slopes, allowing quick access to the application/core pair with the steepest slope.
heapify(S) = heap: This function creates a heap data structure from the slopes in set S.
get_max(heap): This function removes and returns the steepest slope in the heap.
heap_insert(heap, $pp\_slope(a_i, c_i, l_j)$): This function inserts a slope into the heap and recovers the max heap property.

Figure 7.4: Definitions for adaptive thread management algorithms.

A further challenge for scheduling is that there is no simple effective *a priori* way to model the power-performance tradeoffs of running a given application on a particular core. Unlike power management where it can be assumed that performance is linearly related to voltage and power is cubically related (see Section 7.3.3), there is no clear-cut method for estimating the interaction of core heterogeneity and application behavior. Instead, applications must be sampled either offline or online on the actual cores. A machine learning model could be developed to provide insight into the interaction of different application and cores [11], but even such a model must be trained on a set of samples. Consequently, all of the algorithms we present for thread scheduling base their decisions on sampling information obtained during online profiling.

In the following paragraphs, we describe the scheduling algorithms studied in this chapter, including the rationale for each approach, the nature of the sampling required, and an analysis of their computational complexity. We also include a pseudocode implementation of the algorithms where appropriate.

Brute Force: The simplest method for determining the best assignment of threads to cores is to try every possibility and pick the best one. However, this technique suffers from two critical drawbacks when cores can differ due to dynamic heterogeneity. On a chip with *n* cores running *n* applications, there are n! ways of assigning applications to cores. This necessitates taking an infeasible number of samples as the number of cores increases even beyond four cores. If the scheduler assumes that the interactions between applications running on different cores is minimal and ignores them, the algorithm can reduce the sampling to trying every benchmark on every core, for n^2 samples. Since all applications can be sampled on one of the cores during each sampling interval, collecting these n^2 samples requires *n* sampling periods. This approach is analogous to the *sample-one* dynamic scheduling heuristic from Kumar et

al. [68], but for unpredictable rather than designed heterogeneity. While this heuristic greatly reduces the number of samples, because each core could be degraded differently, the scheduler must still compute the overall performance of the n! assignments, leading to an infeasible O(n!) algorithm. Due to the impractical runtime of this algorithm, it is not considered further in the chapter.

Greedy Algorithm: On the other end of the spectrum from brute force, a greedy approach can be employed. Greedy algorithms are popular due to their simple implementation and low runtime complexity. However, they are most effective when solving problems with straightforward solution spaces, such as convex optimization, since greedy solvers typically find local maxima. For this study, we adapt the *VarF&AppIPC* scheduling algorithm from Teodorescu and Torrellas [134], which has been shown to be very effective when combined with global power management on multi-core processors that suffer from process variations (but without manufacturing defects and wear-out faults). This algorithm ranks the applications by average IPC and ranks the cores by inherent frequency (before applying power management) and matches applications and cores by rank in an effort to assign high ILP threads to high frequency cores and memory-bound threads to low frequency cores. Since our cores are heterogeneous, our adaptation of *VarF&AppIPC* samples the IPC of each application on every core and averages the result to obtain an IPC value that can be fairly compared between benchmarks. This requires n^2 samples as in the *sample-one* technique. The complexity of this greedy algorithm is O(n·logn) because the rate determining step is to sort the applications by IPC to determine their rank. (Sorting the cores by frequency can be done offline, since the impact of process variations on frequency can be determined at manufacturing time and the degradation due to wear-out happens over months of use.) Consequently, the Greedy Algorithm executes far

faster than brute force. Below is the pseudocode for our Greedy Scheduling Algorithm:

Input:

(1) The number of cores and applications, n.

(2) The set of applications, A.

(3) The set of cores, C.

(4) The function *freq()* mapping cores, C, and power levels, L, to the frequency range, FR.

Output:

(1) The application scheduling assignment function *sch()* containing the mapping of applications, A, to cores, C, that maximizes throughput.

(2) The total chip throughput, T_{total}, of the best scheduling assignment.

Steps:

1. (*Assign each core to the middle power level*)
   a. For i from 1…n:
      $gpm(c_i) = m$.
   b. End.
2. (*Sample the IPCs of each application on each core and calculate the average IPC of each application across all the cores*)
   For i from 1…n:
   a. $total\_ipc(a_i, m) = 0$.
   b. (*Sample the IPC of this application on all cores and sum the IPCs*)
      For j from 1…n:
      i. $ipc(a_i, c_j, m) = sam\_ipc(a_i, c_j, m)$.
      ii. $total\_ipc(a_i, m) = total\_ipc(a_i, m) + ipc(a_i, c_j, m)$.

End.

c. (*Calculate the application's average IPC*)

avg_ipc(a_i, m) = total_ipc(a_i, m) / n.

End.

3. (*Rank the applications based on IPC*) Reorder the indices of the set of applications, A, such that avg_ipc(a_i, m) ≥ avg_ipc(a_{i+1}, m) for i ranging from 1 to n-1.
4. (*Rank the cores based on nominal frequency*) Reorder the indices of the set of cores, C, such that freq(c_i, p) ≥ freq(c_{i+1}, p) for i ranging from 1 to n-1.
5. (*Go down the ranking assigning applications to the corresponding cores and calculate the total throughput of this scheduling assignment*) $T_{total} = 0$.

   For i from 1…n:

   a. (*Match corresponding applications and cores*) sch(a_i) = c_i.
   b. (*Calculate the throughput of the assignment using formula (7.3-1)*)

      thr(a_i, sch(a_i), m) = ipc(a_i, sch(a_i), m) × freq(sch(a_i), m).
   c. (*Sum up each core's throughput*) $T_{total} = T_{total}$ + thr(a_i, sch(a_i), m).

   End.
6. Return (sch(), T_{total}).

Local Search: For our combinatorial optimization algorithm, we implement the Local Search N/2 algorithm, which was shown to be far more effective than Global Search and Local Search 1 in Section 6.3.1. (We also implemented an advanced combinatorial, non-greedy algorithm called Simulated Evolution [112] but found that the additional performance obtained is insufficient to compensate for the significantly higher implementation complexity and runtime.) However, in this chapter, we optimize for maximum overall throughput rather than energy-delay-squared (ED^2).

Local Search is an archetype for iterative optimization approaches and the basis for many more advanced approaches. While Local Search is greedy by nature, the improvement introduced whereby a solution can be partially accepted offsets much of this limitation. Partially accepting a solution involves retaining any pair-wise swaps that locally improved the performance of the two benchmarks involved and rejecting those swaps which did not, rather than accepting a solution only in full. During each iteration of the algorithm, Local Search selects an assignment among the neighbors of the current best solution and then samples the applications on their assigned cores to determine the performance of this schedule. In our implementation, we run *n* iterations with *n* cores, which means that $O(n^2)$ samples are again required. Furthermore, each iteration does O(n) amount of computation, leading to an overall complexity of $O(n^2)$ for the algorithm. See below for the pseudocode of this algorithm:

Input:

(1) The number of cores and applications, n.

(2) The set of applications, A.

(3) The set of cores, C.

(4) The function *freq()* mapping cores, C, and power levels, L, to the frequency range, FR.

Output:

(1) The application scheduling assignment function *sch()* containing the mapping of applications, A, to cores, C, that maximizes throughput.

(2) The total chip throughput, T_{total}, of the best scheduling assignment.

Steps:

1. (*Assign each core to the middle power level*)
    a. For i from 1…n:

gpm(c_i) = m.

b. End.

2. (*Start the search in some initial random configuration*)

   a. (*Copy the set of cores to a temporary location*) $C_T = C$.

   b. (*Assign the applications to unique cores*)

      For i from 1…n:

      i. (*Select a random core from the set*) c_i = random(C_T).

      ii. (*Assign that core to the application*) sch(a_i) = c_i.

      iii. (*Remove the selected core from the set*) $C_T = C_T \setminus c_i$.

      End.

3. (*Calculate the throughput of each core in the starting assignment*)

   For i from 1…n:

   a. (*Sample each application's IPC on its assigned core*)

      ipc(a_i, sch(a_i), m) = sam_ipc(a_i, sch(a_i), m).

   b. (*Calculate the throughput of the application using formula (7.3-1)*)

      thr(a_i, sch(a_i), m) = ipc(a_i, sch(a_i), m) × freq(sch(a_i), m).

   End.

4. (*Run n iterations of Local Search with n/2 pair-wise swaps*)

   For i from 1…n:

   a. (*Copy the set of applications to a unique location*) $A_T = A$.

   b. (*Create random pairings of all the cores and swap the core assignments of the pairs*)

      For j from 1…n/2:

      i. (*Select a random application from the set*) a_a = random(A_T).

      ii. (*Remove the selected application from the set*) $A_T = A_T \setminus a_a$.

      iii. (*Select another random application from the set*)

a_b = random(A_T).

iv. (*Remove the other selected application from the set*)

$A_T = A_T \setminus a_b$.

v. (*Assign these two cores to pair j*) $pair_j = (a_a, a_b)$.

vi. (*Swap the core assignments*) temp = $sch(a_a)$, $sch(a_a) = sch(a_b)$, $sch(a_b)$ = temp.

End.

c. (*Sample the applications on their newly assigned cores*)

For j from 1…n:

i. (*Sample each application's IPC on its assigned core*)

$ipc(a_j, sch(a_j), m) = sam\_ipc(a_j, sch(a_j), m)$.

ii. (*Calculate the throughput of the application using formula (7.3-1)*)

$thr(a_j, sch(a_j), m) = ipc(a_j, sch(a_j), m) \times freq(sch(a_j), m)$.

End.

d. (*Compare the new throughput values of each pair of application to their old values. If the old values sum to a higher total, switch the cores back to their old assignment*)

For j from 1…n/2:

i. (*Extract the applications from the pair*) $(a_a, a_b) = pair_j$.

ii. (*Compare the old and new throughput values*)

$old\_thr = thr(a_b, sch(a_a), m) + thr(a_a, sch(a_b), m)$

$new\_thr = thr(a_a, sch(a_a), m) + thr(a_b, sch(a_b), m)$

If $new\_thr < old\_thr$ then

(*Switch applications back to old cores*) temp = $sch(a_a)$, $sch(a_a) = sch(a_b)$, $sch(a_b)$ = temp.

End.

End.

5. (*Calculate the total chip throughput*) $T_{total} = 0$.

   For i from 1…n:

   (*Sum up each core's throughput*) $T_{total} = T_{total} + thr(a_i, sch(a_i), m)$.

   End.

6. Return (sch(), T_{total}).

Hungarian Algorithm: Linear programming is a highly general solution method for solving any kind of optimization problem. The key requirement is finding a scheme for converting the constraints and optimization objective of a problem into linear equations or inequalities. While generalized linear programming solvers can be the most effective approach for finding a good solution, certain linear programming problems can be solved more efficiently by exploiting the special structure of the given problem. As explained in Chapter 4, OS scheduling can be modeled as the classic Assignment Problem from operations research. For this problem where a solution is desired with a one-to-one mapping of applications to cores, the Hungarian Algorithm [93] is the standard solution approach. One clear advantage of the Hungarian Algorithm over the simplex method (generally considered the most effective algorithm for general linear programming) is that the Hungarian Algorithm has a bounded worst-case runtime of $O(n^3)$ for a processor with *n* cores [25]. On the other hand, the simplex method has an exponential runtime in the worst case and a polynomial-time average case complexity that is highly dependent on the nature of the objective function and constraints [49]. We implement the $O(n^3)$ Hungarian Algorithm described in [25], as well as the $O(n^4)$ Munkres version [93] described by Pilgrim [97] that was employed in Section 4.2.2, and show that an efficient implementation is

crucial. As with brute force and the Greedy Scheduling Algorithm, the Hungarian Algorithm must sample each application on each core, requiring *n* sampling intervals, to create a matrix of the benefit of assigning each application to each core before running the actual algorithm. A clear advantage of the Hungarian Algorithm over the above approaches is that it finds the optimal solution, provided that the assumption of negligible interference between applications holds and that the samples accurately reflect thread behavior.

Hierarchical Hungarian Algorithm: The above four scheduling algorithms suffer from two main drawbacks. First, each algorithm requires *n* sampling intervals to provide the necessary performance evaluation of the different thread-core matchings. In future many-core processors, this will require hundreds of sampling intervals. These sampling intervals must be of reasonable length in order to ensure that they are reflective of the actual application behavior. In our experimental work, we found that sample lengths must be on the order of million of cycles to amortize the impact of context switching and cache and branch predictor warm-up. However, running hundreds of million cycle samples is impractical because it would mean that most or even all of the time between scheduling intervals would be consumed just with the samples. Furthermore, even if the time to collect hundreds of samples was available, by the time the samples are collected – hundreds of millions or billions of cycles later – a good number of the applications would have changed phases thereby invalidating the samples. This is a significant challenge of many-core processors that does not become apparent until hundreds of cores are put on the die.

The second drawback is the time complexity of most of the above algorithms. Clearly brute force, with exponential runtime, is infeasible. Nonetheless, as our experimental results will demonstrate, even an $O(n^2)$ or $O(n^3)$ algorithm can become too time consuming to perform at a desirable scheduling interval granularity. For these

two reasons, an alternative scheduling algorithm is needed for future many-core systems.

As the Hungarian Algorithm is the most effective of the above techniques (Section 7.5.2), we create a Hierarchical Hungarian Algorithm that requires significantly fewer samples and a far shorter runtime than the previously proposed methods. This algorithm divides the cores of the CMP into *g* groups of 8, 16, or 32 cores (experimentally determined to be reasonable sizes) and obtains a locally effective scheduling assignment within each group. Rather than sampling all threads on all cores, applications are only sampled on those cores in their group and the solution to this much smaller assignment problem is computed using the Hungarian Algorithm. The resulting algorithm must collect $(n/g)^2$ samples in n/g intervals and run the Hungarian Algorithm on *g* smaller problems of size n/g for a total time complexity of $O(n^3/g^2)$. If the number of groups is a function of the number of cores in the processor, say $g = n / s$, the size of each group will be a fixed size, $s = n / g$, leading to a complexity of $O(n \cdot s^2)$, effectively linear time complexity. Figure 7.5 presents some definitions specific to the Hierarchical Hungarian Algorithm.

hungarian(A, C, cost()) = ass(): This function executes the Hungarian Algorithm which finds the lowest cost assignment (ass()) of applications in set A to cores in set C using the cost matrix and returns a function mapping applications to cores. cost(i, j) returns the cost of assigning application, a_i, to core, c_j. The function $ass(a_i) = c_i$ takes in an application, a_i, and returns the core, c_i, that the application is assigned to in the minimum cost assignment.
s: The size of the groups used in the Hierarchical Hungarian Scheduling Algorithm.
a_{ij}: This represents the j^{th} application of group i.
c_{ij}: This represents the j^{th} core of group i.
cost(i, j): This function represents the cost of running the i^{th} application on the j^{th} core.

Figure 7.5: Definitions for the Hierarchical Hungarian Scheduling Algorithm.

Below is the pseudocode for the Hierarchical Hungarian Scheduling Algorithm:

Input:

(1) The number of cores and applications, n.

(2) The set of applications, A.

(3) The set of cores, C.

(4) An initial mapping *init_sch()* of applications, A, to cores, C,

(5) The function *freq()* mapping cores, C, and power levels, L, to the frequency range, FR.

(6) The size of the hierarchical groups, s.

Output:

(1) The application scheduling assignment function *sch()* containing the scheduling assignment mapping applications, A, to cores, C, which maximizes throughput.

(2) The total chip throughput, T_{total}, of the best scheduling assignment.

Steps:

1. (*Assign each core to the middle power level*)
   a. For i from 1…n
      $gpm(c_i) = m$.
   b. End.
2. (*Randomly partition the applications and cores into groups of size s*)
   a. (*Copy the set of applications to a temporary location*) $A_T = A$.
   b. (*Assign the application to groups*)
      For i from 1…n/s:
         For j from 1…s:
            i. (*Select random application for the group*) $a_{ij} = random(A_T)$.
            ii. (*Add this application to the set for the group*) $A_i = A_i \cap a_{ij}$
            iii. (*Assign that application's core to the group*) $c_{ij} = init\_sch(a_{ij})$
            iv. (*Add this core to the set for the group*) $C_i = C_i \cap c_{ij}$
            v. (*Remove the selected application from the set*) $A_T = A_T \setminus a_{ij}$.

End.

End.

3. (*Run the Hungarian Algorithm on each group of applications and cores*)

   (*Iterate through each group*)

   For i from 1…n/s:

   a. (*Sample the performance of the group's applications of each its cores*)

      (*Iterate through each application in the group*)

      For j from 1…s:

         (*Iterate through each core in the group*)

         For k from 1…s:

            i. (*Sample the application's IPC*)

               $ipc(a_{ij}, c_{ik}, m) = sam\_ipc(a_{ij}, c_{ik}, m)$.

            ii. (*Calculate the throughput of the application using formula (7.3-1)*)

               $thr(a_{ij}, c_{ik}, m) = ipc(a_{ij}, c_{ik}, m) \times freq(c_{ik}, m)$.

         End.

      End.

   b. (*Convert the problem to a minimization problem by creating a cost matrix*)

      i. (*Find the largest throughput value in this group*) largest = 0.

         (*Iterate through each application in the group*)

         For j from 1…s:

            (*Iterate through each core in the group*)

            For k from 1…s:

               If $thr(a_{ij}, c_{ik}, m) >$ largest, largest = $thr(a_{ij}, c_{ik}, m)$.

            End.

End.

ii. (*Create a cost matrix by subtracting each throughput value from the largest*)

(*Iterate through each application in the group*)

For j from 1…s:

(*Iterate through each core in the group*)

For k from j…s:

cost(j, k) = largest - thr(a_{ij}, c_{ik}, m).

End.

End.

c. (*Run the Hungarian Algorithm on the group to get the best assignment*)

ass_i = hungarian(A_i, C_i, cost()).

4. (*Amalgamate the scheduling assignments of each group and calculate the chip's total throughput*) $T_{total} = 0$.

(*Iterate through the groups*)

For i from 1…n/s:

(*Iterate through the applications in group i*)

For j from 1…s:

(*Add the core assignment of the j^{th} application of group i to the overall scheduling assignment for the processor*)

sch(a_{ij}) = ass_i(a_{ij}).

(*Sum up each core's throughput*) $T_{total} = T_{total}$ + thr(a_{ij}, sch(a_{ij}), m).

End.

End.

5. Return (sch(), T_{total}).

The Hierarchical Hungarian Algorithm has multiple advantages over the other scheduling techniques. The number of samples is reduced to a more manageable number of 8 to 32, which allows scheduling to be run more frequently and samples to be longer and more accurate. In addition, the runtime of the algorithm is much shorter since the problem size is reduced from considering hundreds of applications and cores to no more than 32. Finally, since each group can be sampled simultaneously and the assignment problems solved independently, each group's sub-problem can be solved in parallel. This reduces the algorithm's complexity to $O(s^3)$, making it completely independent of the size of the many-core processor. We show in our results that parallelizing the Hierarchical Hungarian Algorithm this way provides a significant runtime speedup because (1) the computation is overlapped and (2) the sequential hierarchical solver must deal with a larger data set (for the whole chip), whereas the much smaller data footprint of the sub-problem greatly reduces the processing time of the algorithm.

One concern of this hierarchical approach is that the locally optimal solution may not be close to the globally optimal schedule, because applications are limited to finding a suitable core within their group. We address this possibility by randomizing the grouping of the applications and cores in each scheduling quantum, allowing applications to be sampled and assigned to different cores each time the scheduler is called. In theory, over time, each application will have the opportunity to run on each core and the Hierarchical Hungarian Algorithm's schedule and performance will converge to the standard Hungarian Algorithm's schedule over a number of scheduling intervals. The impact of adding randomized groupings will be investigated in Section 7.5.4.

Table 7.1 provides an overview of the features of the scheduling algorithms that we explore.

Table 7.1: A summary of the features of the application scheduling algorithms.

Scheduling Algorithm	Computational Complexity	Required Number of Sampling Intervals
Brute Force	O(n!)	n
Greedy Algorithm (VarF&AppIPC)	O(n·logn)	n
Local Search (n/2 swaps)	$O(n^2)$	n
Hungarian Algorithm	$O(n^3)$	n
Sequential Hierarchical Hungarian Algorithm	$O(n^3/g^2) \rightarrow O(n \cdot s^2)$	$n/g \rightarrow s$
Parallel Hierarchical Hungarian Algorithm	$O(s^3)$	s

7.3.3. Global Power Management Algorithms

In addition to developing scalable application scheduling algorithms, this chapter conducts the first study of global power management (GPM) for many-core architectures suffering from manufacturing defects and lifetime wear-out, in addition to process variations. As per prior work [10][53][58][88][119][134], the objective is to maximize throughput under a chip-wide power budget. In this study, we focus on dynamic voltage and frequency scaling (DVFS), the most widely implemented GPM approach. Like [10][53][58][88][119][134], we assume that each core has independent frequency and voltage control. Given the complexity of tens or hundreds of voltage and frequency domains on the chip, future many-core processors might group multiple cores into one voltage/frequency island [53]. However, fixed sized groups will only provide temporary relief for the complexity of the power management problem, so long as the industry continues to double transistor density each generation.

We consider a DVFS mechanism that scales frequency linearly with voltage (as per prior work) and has seven discrete voltage levels spaced out evenly between 0.7V and 1.0V (the nominal voltage). The corresponding frequency range is dependent on the impact of process variations on a given core (as described above) but would vary from 2.8 GHz to 4.0 GHz (the nominal frequency) on a core unaffected by variations.

One distinct difference between GPM and scheduling is that the impact of changing the voltage and frequency of a core on application performance and power dissipation can be estimated effectively by simply knowing the power-performance characteristics of the application on the core at the current DVFS level. As in Chapter 5, we employ the model of Isci et al. [58] described in Figure 5.2, and assume that within the narrow range of DVFS levels, performance is linearly proportional to voltage and power is a cubic function of voltage. Consequently, GPM algorithms using this relationship need only one sample of each application on its assigned core.

The above model has a couple of limitations. As discussed in Meng et al. [88], it neglects to account for memory-bound applications that do not speed up effectively with increased frequencies or likewise slow down as fast with decreasing frequencies. However, we found experimentally that this issue was not a great cause for concern among our power management algorithms. On the other hand, we found that the assumed cubic dependence of power on voltage was very conservative, often leading to DVFS level assignments under the chip-wide budget. To a great extent, this can be attributed to the influence of leakage power, which has a less than cubic dependence on voltage. In our experiments, we empirically explore models that assume power is proportional to voltage raised to the power of 3, 2.75, and 2.5 and find that performance can be increased while still meeting the global power budget with the use of the 2.5 power proportionality between voltage and power. See formula 3 in Figure 7.3.

As with the scheduling algorithms, we examine five approaches to power management: a brute force method, a greedy algorithm, a heuristic combinatorial optimization scheme, linear programming, and a hierarchical approach. In the following paragraphs, we discuss these algorithms, their sampling requirements, and their computational complexity.

Brute Force: Isci et al. propose the MaxBIPS algorithm [58], which uses the model discussed above to calculate the performance and power dissipation achieved for each combination of power settings available on the chip. Assuming that DVFS can be set to p discrete levels (for Isci et al., $p = 3$ and for our work $p = 7$), there are p^n possible power settings for a CMP with n cores. For each power setting option, the calculated performance and power of each core must be summed to determine the chip throughput and power, requiring O(n) time. While MaxBIPS is very effective at calculating a good DVFS assignment with a single sample per application/core at an isometric voltage setting, clearly even for $p = 3$, the $O(n \cdot p^n)$ computation cost is prohibitive for many-core processors. Consequently, we do not consider MaxBIPS further in this chapter.

Greedy Algorithm: We develop a simple greedy approach to power management which leverages a key intuition about global power management for maximum throughput. Essentially, performance is maximized by shifting power to applications which can individually generate the highest throughput. To achieve this, the greedy algorithm gives as much power as possible to application/core combinations with the greatest inherent performance capability. As with MaxBIPS, a single sample is taken for each scheduler-assigned application/core pair at an isometric voltage and frequency setting (the middle level) and the throughput in billions of instructions per second (BIPS) is calculated. The BIPS value is a function of both the application IPC on the assigned degraded core and the core's operating frequency due to variations. Each core is then set to the lowest DVFS setting and the voltage/performance/power model is used to calculate the power for the lowest power configuration, which is subtracted from the power available to meet the budget. The pairs are then ranked by this BIPS measurement and, starting with the highest ranked pair, cores are greedily set to the highest voltage/frequency setting proceeding down the ranking until,

according to the voltage/performance/power model, the power budget is reached. If there is some leftover power that was insufficient to allow the final core to be set to the highest setting, that core is set to the highest setting meeting the budget. The rest of the lower ranked cores are then left at the lowest DVFS setting. Since the most complex step of the greedy algorithm is ranking the application/core pairs by BIPS, the algorithm's complexity is O(n·logn). See below for a pseudocode implementation of this algorithm:

Input:

(1) The number of cores and applications, n.

(2) A scheduling assignment *sch()* matching the applications, A, to the cores, C.

(3) The function *volt()* mapping power levels, L, to the voltage range, VR.

(4) The function *freq()* mapping cores, C, and power levels, L, to the frequency range, FR.

(5) The global power budget P_{max}.

Output:

(1) The global power management assignment function *gpm()* assigning power levels, L, to cores, C, to maximize overall throughput.

(2) The total estimated throughput, $T_{total,}$ of the best global power management assignment.

(3) The total estimated power dissipation, $P_{total,}$ for the best GPM assignment.

Steps:

1. (*Sample the IPCs and power of the applications and cores at the middle power level, m*)

   For i from 1…n:

   $ipc(a_i, sch(a_i), m) = sam\_ipc(a_i, sch(a_i), m)$.

$pow(a_i, sch(a_i), m) = sam\_ pow(a_i, sch(a_i), m)$.

End.

2. (*Calculate the throughput of each core from the sampled IPCs and the middle level frequencies of each core using formula (7.3-1)*)

   For i from 1…n:

   $thr(a_i, sch(a_i), m) = ipc(a_i, sch(a_i), m) \times freq(sch(a_i), m)$.

   end.

3. (*Set the available power to the total power budget*) $P_{avail} = P_{max}$.

4. (*Initialize the global power management assignment by setting each core to the lowest power setting*)

   For i from 1…n:

   a. (*Set the core to the lowest power level*) $gpm(sch(a_i)) = 1$.

   b. (*Calculate the estimated power consumed at that level using formula (7.3-3)*)

      $est\_pow(a_i, sch(a_i), 1) = pow(a_i, sch(a_i), m) \times volt(1)^r / volt(m)^r$

   c. (*Subtract this minimum power from the available power*)

      $P_{avail} = P_{avail} - est\_pow(a_i, sch(a_i), 1)$

   End

5. (*Rank the applications based on throughput*) Reorder the indices of the set of applications, A, such that $thr(a_i, sch(a_i), m) \geq thr(a_{i+1}, sch(a_{i+1}), m)$ for i ranging from 1 to n-1.

6. (*Go down the ranking assigning full power to as many cores as possible*)

   For i from 1…n:

   a. (*Calculate the estimated increased power of this application at the highest power level using formula (7.3-3)*)

      $est\_pow(a_i, sch(a_i), p) = pow(a_i, sch(a_i), m) \times volt(p)^r / volt(m)^r$

b. (*If there is enough available power, set this core to the highest power setting, and adjust the available power accordingly*)

If (est_pow(a_i, sch(a_i), p) - est_pow(a_i, sch(a_i), 1)) ≤ $P_{avail,}$

gpm(sch(a_i)) = p.

P_{avail} = P_{avail} - (est_pow(a_i, sch(a_i), p) - est_pow(a_i, sch(a_i), 1)).

End.

c. Otherwise, set last = i and exit the for loop.

End.

7. (*Set the application with index 'last' to the highest power setting possible to use up the remaining available power*)

For j from p-1…2:

a. (*Calculate the estimated power this application at power level j using formula (7.3-3)*)

est_pow(a_{last}, sch(a_{last}), j) = pow(a_{last}, sch(a_{last}), m) × volt(j)^r / volt(m)^r

b. (*If there is enough available power, set the core to this power level*)

If ((est_pow(a_{last}, sch(a_{last}), j) - est_pow(a_i, sch(a_i), 1)) ≤ P_{avail},

gpm(sch(a_{last})) = j.

Exit loop.

End.

End.

8. (*Calculate the total estimated chip throughput and power dissipation*)

T_{total} = 0, P_{total} = 0.

For i = 1…n

a. (*Sum up each core's throughput*)

T_{total} = T_{total} + thr(a_i, sch(a_i), m) × volt(gpm(sch(a_i))) / volt(m).

b. (*Sum up each core's power*)

$P_{total} = P_{total} + est\_pow(a_i, sch(a_i), gpm(sch(a_i)))$.

End.

9. Return (gpm(), T_{total}, P_{total}).

Steepest Descent: Our heuristic optimization algorithm is Steepest Descent, which is essentially directed Local Search. Rather than randomly select a configuration in the neighborhood of the current best known configuration as is done in Local Search for thread scheduling, Steepest Descent exploits the known correlation between voltage, performance, and power to direct the search. We employ the algorithm from Meng et al. [88] that was designed to address the large search space resulting from applying multiple power optimizations simultaneously. In our work, we only use DVFS, but because of the large scale of our many-core architecture, the optimization problem is sufficiently challenging with DVFS alone. As with MaxBIPS and the Greedy Algorithm, each application is sampled on its assigned core at the middle DVFS level to calibrate the voltage/performance/power model.

The algorithm starts by assuming each core is set to the highest power setting. Then using the analytical model, if the power is estimated to be over the chip-wide budget, the algorithm selects the application/core pair that would provide the biggest ratio of power reduction for performance loss if the voltage was dropped one step. This new configuration's power dissipation is estimated and, if the power is still over budget, the steepest descent is again calculated from the new configuration. This process is repeated until the power budget is met. To optimize the runtime of Steepest Descent, we use a max-heap data structure for storing the ranking of the power reduction to performance loss ratio for each application/core pair. In the worst case, the Steepest Descent Algorithm would have dropped the voltage/frequency settings from the highest values all the way to the lowest for each core. This would involve *n* x

p iterations for *n* cores and *p* power levels. Using the efficient heap data structure, accessing the steepest drop and updating the data structure at each iteration takes O(log n) time, for a total complexity of O(p·n·log n). Below is a pseudocode description of the algorithm:

Input:

(1) The number of cores and applications, n.

(2) A scheduling assignment *sch()* matching the applications, A, to the cores, C.

(3) The function *volt()* mapping power level, L, to the voltage range, VR.

(4) The function *freq()* mapping cores, C, and power levels, L, to the frequency range, FR.

(5) The global power budget P_{max}.

Output:

(1) The global power management assignment function *gpm()* assigning power levels, L, to cores, C, to maximize overall throughput.

(2) The total estimated throughput, $T_{total,}$ of the best global power management assignment.

(3) The total estimated power dissipation, $P_{total,}$ for the best GPM assignment.

Steps:

1. (*Sample the IPCs and power of the applications and cores at the middle power level, m*)

   For i from 1…n:

   $ipc(a_i, sch(a_i), m) = sam\_ipc(a_i, sch(a_i), m)$.

   $pow(a_i, sch(a_i), m) = sam\_ pow(a_i, sch(a_i), m)$.

   End.

2. (*Calculate the throughput of each core from the sampled IPCs and the middle level frequencies of each core using formula (7.3-1)*)

   For i from 1…n:

   $thr(a_i, sch(a_i), m) = ipc(a_i, sch(a_i), m) \times freq(sch(a_i), m)$.

   End.

3. (*Set the starting power level to the highest level. Calculate the estimated power and throughput of the cores at this level and the total chip power*) $P_{total} = 0$.

   For i from 1…n:

   a. $gpm(sch(a_i)) = p$.
   b. (*Calculate the estimated throughput at this level using formula (7.3-2)*) $est\_thr(a_i, sch(a_i), p) = thr(a_i, sch(a_i), m) \times volt(p) / volt(m)$
   c. (*Calculate the estimated power consumed at this level using formula (7.3-3)*) $est\_pow(a_i, sch(a_i), p) = pow(a_i, sch(a_i), m) \times volt(p)^r / volt(m)^r$
   d. (*Sum up each core's power*) $P_{total} = P_{total} + est\_pow(a_i, sch(a_i), p)$.

   End.

4. (*If the starting total power is under the power budget, the algorithm is completed. Return the power level assignment, the total throughput and total power*)

   If $P_{total} \leq P_{max}$

   a. (*Calculate the total throughput of the chip*) $T_{total} = 0$.

      For i from 1…n:

      (*Sum up each core's throughput*)

      $T_{total} = T_{total} + est\_thr(a_i, sch(a_i), p)$.

      End.

   b. Return $(gpm(), T_{total}, P_{total})$.

End.

5. (*Otherwise, calculate the power-performance slope of each application on its assigned core when set to the highest power level and collect the slopes in a set*) $S = \emptyset$.

   For i from 1…n:

   a. (*Calculate the estimated throughput at level p - 1 using formula (7.3-2)*) $est\_thr(a_i, sch(a_i), p - 1) = thr(a_i, sch(a_i), m) \times volt(p - 1) / volt(m)$.

   b. (*Calculate the estimated power at level p - 1 using formula (7.3-3)*) $est\_pow(a_i, sch(a_i), p - 1) = pow(a_i, sch(a_i), m) \times volt(p - 1)^r / volt(m)^r$.

   c. (*Calculate the power-performance slope using formula (7.3-4)*) $pp\_slope(a_i, sch(a_i), p) = (est\_pow(a_i, sch(a_i), p) - est\_pow(a_i, sch(a_i), p - 1)) / (est\_thr(a_i, sch(a_i), p) - est\_thr(a_i, sch(a_i), p - 1))$.

   d. (*Add the slope to the set*) $S = S \cup pp\_slope(a_i, sch(a_i), p)$.

   End.

6. (*Create a max-heap from the set of power-performance slopes*) heap = heapify(S).

7. (*Run the main loop where the search space is explored along the steepest power-performance gradient to find a good power setting assignment*)

   While $P_{total} > P_{max}$:

   a. (*Get the steepest slope out of the heap*) $pp\_slope(a_i, sch(a_i), l_j) = get\_max(heap)$.

   b. (*Reduce this application/core pair's power level by one*). $gpm(sch(a_i)) = l_j - 1$.

   c. (*Adjust the estimated chip total power correspondingly*) $P_{total} = P_{total} - est\_pow(a_i, sch(a_i), l_j) + est\_pow(a_i, sch(a_i), l_j - 1)$.

d. (*If the core is not yet at the lowest power level, recalculate and insert its updated power-performance slope into the heap*)

If $l_j - 1 > 1$:

i. (*Estimate the throughput at level l_j - 2 using formula (7.3-2)*) $est\_thr(a_i, sch(a_i), l_j - 2) = thr(a_i, sch(a_i), m) \times volt(l_j - 2) / volt(m)$.

ii. (*Estimate the power consumption at level l_j - 2 using formula (7.3-3)*) $est\_pow(a_i, sch(a_i), l_j - 2) = pow(a_i, sch(a_i), m) \times volt(l_j - 2)^r / volt(m)^r$.

iii. (*Calculate the power-performance slope using formula (7.3-4)*) $pp\_slope(a_i, sch(a_i), l_j - 1) = (est\_pow(a_i, sch(a_i), l_j - 1) - est\_pow(a_i, sch(a_i), l_j - 2)) / (est\_thr(a_i, sch(a_i), l_j - 1) - est\_thr(a_i, sch(a_i), l_j - 2))$.

iv. $heap\_insert(heap, pp\_slope(a_i, sch(a_i), l_j - 1))$.

End.

End.

8. (*Calculate the total estimated chip throughput and power dissipation*)

$T_{total} = 0$, $P_{total} = 0$.

For i = 1…n

a. (*Sum up each core's throughput*)

$T_{total} = T_{total} + est\_thr(a_i, sch(a_i), gpm(sch(a_i)))$.

b. (*Sum up each core's power*)

$P_{total} = P_{total} + est\_pow(a_i, sch(a_i), gpm(sch(a_i)))$.

End.

9. Return $(gpm(), T_{total}, P_{total})$.

LinOpt: Teodorescu and Torrellas [134] propose using linear optimization to solve the global power management problem in a multi-core chip afflicted with process variations. Their algorithm, LinOpt, involves three steps. First, the power management task is formulated as a linear programming problem as described in Figure 7.6. Then, the formulation is run through a linear programming solver that implements the simplex method. Linear programming requires continuous-valued variables, and thus the linear solver can return voltage settings that lie between the discrete DVFS levels. Thus, the third step conservatively drops any voltage values to the next lowest DVFS setting. As in all the other algorithms, performance is modeled as linearly dependent on voltage. However, the cubic relationship between voltage and power cannot be captured in a linear program. Instead, a linear approximation is found that minimizes the error with the true relationship as determined by three samples taken at the lowest, middle, and highest DVFS setting [134]. We implement this linear approximation using linear least squares fitting (LLSF), also known as linear regression [101].

Linear regression calculates the slope and intercept of the line that most closely matches the empirically measured voltage/power relationship. These values are then used to formulate global power management as a linear program. Linear regression with three data points can be implemented in O(1) time. Since, this must be done for each application/core combination, the total time is O(n). As mentioned above, the simplex method has exponential worst case complexity and polynomial time average case complexity and thus dominates LinOpt's runtime. Experimental and stochastic analysis [49] have concluded that average case runtime estimates for linear programming are $O(n^4)$ when considering problems where the number of constraints is of the same order as the number of variables, such as in our case. (The power management formulation has 2n+1 constraints including a lower and upper bound for each voltage setting and the chip-wide power budget.) In our results, we evaluate

whether this high-order polynomial runtime becomes a problem for many-core processors.

Linearization Assumptions: $t_i = a_i v_i$; $p_i = m_i v_i + b_i$;

where a_i is determined by sampling and m_i (slope) and b_i (intercept) are determined by sampling followed by linear least squares fitting.

Maximize: Total Throughput $= \sum_{i=1}^{n} a_i v_i$

Subject to: $\forall i$: $v_{min} \leq v_i \leq v_{max}$

$\sum_{i=1}^{n} m_i v_i + b_i \leq P_{max}$

Figure 7.6: The formulation of global power management as a linear program.

In our experiments we noted a consistent problem with the linearization of the voltage/power relationship using an approximation based on the linear regression of the power values obtained at the lowest, middle, and highest DVFS settings as prescribed in [134]. In solving the linear program, the simplex solver works by searching among the extreme points (or corners) of the feasible solution space [49]. The points will set all cores to either the highest or lowest voltage setting, except for at most one. However, as shown in Figure 7.7, it is at these two end points of the voltage range that the linear approximation most underestimates the power dissipation. Consequently, LinOpt often provides a set of DVFS setting which lead to a total power output exceeding the chip-wide budget, which would be unacceptable in a real implementation. We develop a variant on LinOpt (LinOpt2) which corrects this underestimation by using only two power samples – at the lowest and highest voltage settings – to linearize the voltage/power relationship. Instead of using linear least squares fitting, our method calculates the slope and intercept of the line connecting these two sampled points for each application/core assignment, as shown in Figure 7.7. This method has the benefit of being most accurate at the end points of the voltage

range, which are the exact points most used in LinOpt's solution. We will present the results of LinOpt2 in comparison to the original LinOpt in Section 7.5.3.

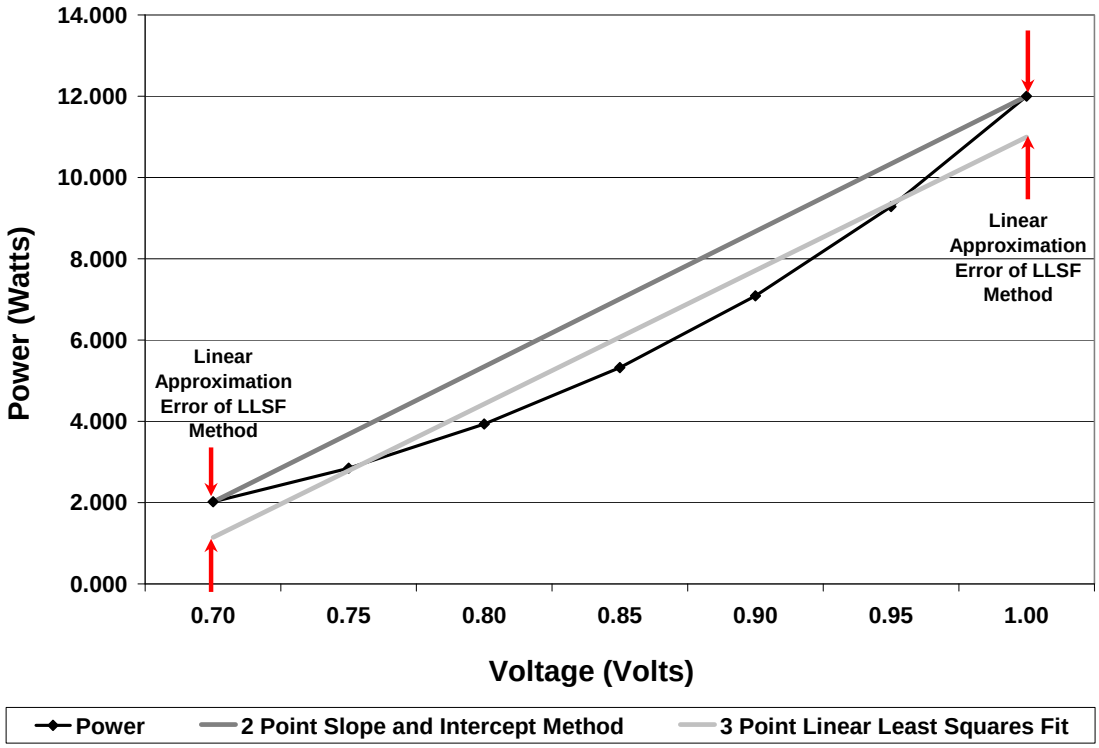


Figure 7.7: Linear approximation methods for the voltage/power relationship.

A possible improvement to linear optimization is to employ integer linear programming (ILP). ILP works with discrete-valued variables and thus is a better match for DVFS settings. This eliminates the need for the third step in LinOpt and can lead to less conservative power settings. We implemented and experimented with a version of LinOpt using ILP. However, our simulation results show only a small performance improvement from this enhanced algorithm. More critically, integer linear programming is a known NP-hard problem (it has no known polynomial time solution) and our experimental results show that the runtime of ILP LinOpt is orders of magnitude slower than LinOpt, giving it poor scalability.

Hierarchical Algorithms: A logical direction to pursue would be to design hierarchical algorithms for GPM to increase scalability in an analogous manner to the Hierarchical Hungarian Algorithm. For instance, a Hierarchical LinOpt Algorithm would divide the chip into *g* groups that are given 1/g of the chip-wide power budget and then solve each resulting sub-problem by applying linear programming. However, our results in Section 7.5.3 show that Steepest Descent is sufficiently effective at finding a power-performance efficient power setting and is highly scalable from a computational complexity perspective, obviating the need for pursuing a hierarchical approach.

Table 7.2 summarizes the major characteristics of the GPM algorithms.

Table 7.2: A summary of the features of the global power management algorithms.

Global Power Management Algorithm	Computational Complexity	Required Number of Sampling Intervals
Brute Force (MaxBIPS)	$O(n \cdot p^n)$	1
Greedy Algorithm	$O(n \cdot \log n)$	1
Steepest Descent	$O(p \cdot n \cdot \log n)$	1
LinOpt	$O(n^4)$ (average case)	3
LinOpt2	$O(n^4)$ (average case)	2

7.4. Methodology

7.4.1. Simulation Infrastructure

A key challenge of this study is developing an experimental infrastructure capable of simulating large-scale many-core processors. In this study, we require hundreds of thousands of microarchitectural simulations. Thus we leverage the hierarchical and parallel simulation infrastructure described in Section 4.3, augmented to support many-core architectures with up to 256 cores, maximum throughput statistics instead of ED^2, and all the new scheduling and power management

algorithms discussed in the prior sections of this chapter. In this study, our baseline core, unaffected by process variations and errors, is a single-threaded, four-way superscalar, out-of-order processor. The core architectural parameters are the same as those for Chapter 5 and are listed in Table 5.2.

The goal of this study is to consider large-scale many-core architectures as well as to thoroughly evaluate the time complexity and effectiveness of a number of scheduling and power management algorithms. Simulating processors with hundreds of cores takes immense amount of computer resources. To make our comprehensive study more tractable, we conduct some of our evaluations in an offline format as in Chapter 6. The use of offline simulation allows us to obtain the oracle results of Section 7.5.1 and well as the 128 and 256 core results of Sections 7.5.2 and 7.5.3 which would have otherwise required weeks or even months of simulation time.

We validate our offline simulations with online runs (Section 7.5.4) that run much longer simulation intervals and advance the applications through their execution to simulate phase changes and dynamic behavior. For the online runs, as discussed above, we use 100ms quanta for application scheduling and 10ms quanta for global power management. At the processor's nominal frequency of 4GHz, this amounts to 400 million cycles and 40 million cycles per scheduling and power management quanta respectively. The 10% of a quantum devoted to sampling is divided equally between the sampling intervals. For example, the sampling period of the scheduler is 40 million cycles, so that with 64 cores, the Hungarian Algorithm would run 64 samples of 625,000 cycles each.

7.4.2. Simulating Unpredictably Heterogeneous Many-Core Processors

We evaluate many-core architectures with four to 256 cores, for which we consider the same three types of degradation as in prior chapters. In this study, a processor core can have at most one to two faults. Cores also have variable

frequencies that are randomly assigned to be in the range of 60% to 110% of the nominal frequency of 4GHz with a bias towards slower cores. Table 7.3 presents a list of the core degradations possible for each type of fault. A degraded many-core processor of the appropriate size is generated by randomly picking a degradation (including no damage) from each of the three types for each core on the chip.

Table 7.3: Possible forms of core degradation.

Type of Degradation	List of Options
Degraded Component	none memory latency is doubled half the L2 cache is broken half the L1 instruction cache is broken a way of the L1 instruction cache is broken front-end bandwidth is reduced from 4-way to 3-way half the integer issue queue is broken integer issue bandwidth is reduced by one one or more integer ALUs are disabled half the rename registers are broken half the load queue is broken half the store queue is broken half the L1 data cache is broken a way of the L1 data cache is broken half the re-order buffer is broken
Frequency Degradation	60 – 110 % of the nominal, set at intervals of 2.5%
Increased Leakage	none 2X nominal in L1 caches and TLBs 2X nominal in front end and re-order buffer 2X nominal in integer back end 2X nominal in floating point back end 2X nominal in load and store queues 2X nominal across core (excluding L2 cache)

7.4.3. Workloads

We randomly generate workloads from among the 17 SPEC CPU 2000 benchmarks for each many-core processor configuration. We use three fast-forward points (one, two, and three billion instructions) for each benchmark to add further diversity to the workloads. In the main scalability study of Sections 7.5.2 and 7.5.3, degraded many-core configurations are constructed from 50 randomly generated cores that run workloads consisting of the above 51 benchmark/fast-forward point

combinations. Given seven DVFS voltage/frequency levels, 17,850 single-core SESC simulations were run for the offline study.

7.4.4. Assessing Algorithm Runtimes

A novel component of our work is our more rigorous analysis of scheduling and power management algorithm runtimes. In addition to the computational complexity results presented in Sections 7.3.2 and 7.3.3, we empirically assess the execution requirements of our algorithms through simulation. Each of the algorithms is implemented in C and compiled with full optimizations into a special MIPS binary that can be executed on SESC. This binary is then run on SESC while modeling the architecture of cores found in our many-core organizations to extrapolate how these algorithms will perform in future processors.

7.5. Results and Discussion

7.5.1. Coordinating Application Scheduling and Global Power Management

In this section we investigate the importance of coordinating application scheduling and power management in future many-core architectures. Section 7.2.2 provides an analytical argument as to why scheduling and power management affect different components of the performance equation. Rather than trying to develop an effective coordinated scheme, we take a more comprehensive approach and compare the performance and power dissipation of a number of approaches which independently schedule and manage power against an oracle policy. If these independent approaches can achieve results very close to that of the optimal scheduler and power manager, then there is no point in even considering a coordinated technique. In every quantum, our oracle algorithm employs a brute force examination of all possible scheduling and power management combinations and then selects the application-to-core assignment and DVFS settings that provide the maximum possible

performance while staying within the chip-wide power budget. Obtaining this optimal solution is simple to implement using our offline infrastructure because we have already executed architectural simulations for all possible assignments of applications to degraded cores at each setting of voltages and frequencies, and only need to iterate through every option.

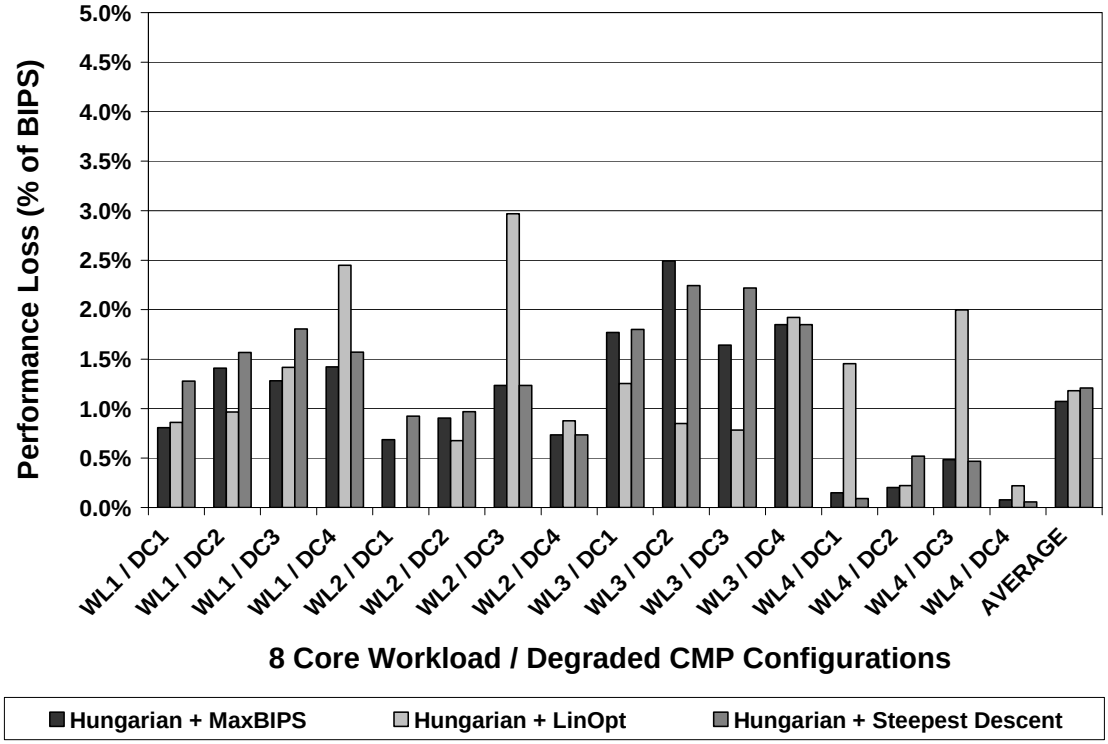


Figure 7.8: Performance loss of uncoordinated scheduling and global power management algorithms relative to the oracle manager.

We compare the performance of this oracle against three combinations of scheduling and power management algorithms (Hungarian coupled with MaxBIPS, LinOpt, and Steepest Descent), chosen on the basis of the results of previous chapters and their superiority in prior work [58][88][134]. In all three combinations, the scheduler and power manager operate independently, but power management is employed after scheduling. This is required to ensure that the chosen schedule does not lead to power overshoots. We verified experimentally that running power

management first, and then scheduling leads to less optimal power-performance efficiency and does indeed create over-budget scenarios. Consequently, our runtime management system always sets the DVFS level of each core to the midpoint during the scheduler sampling period, calculates the best schedule, and then employs global power management during the scheduler's steady phase to ensure that the power budget is not exceeded.

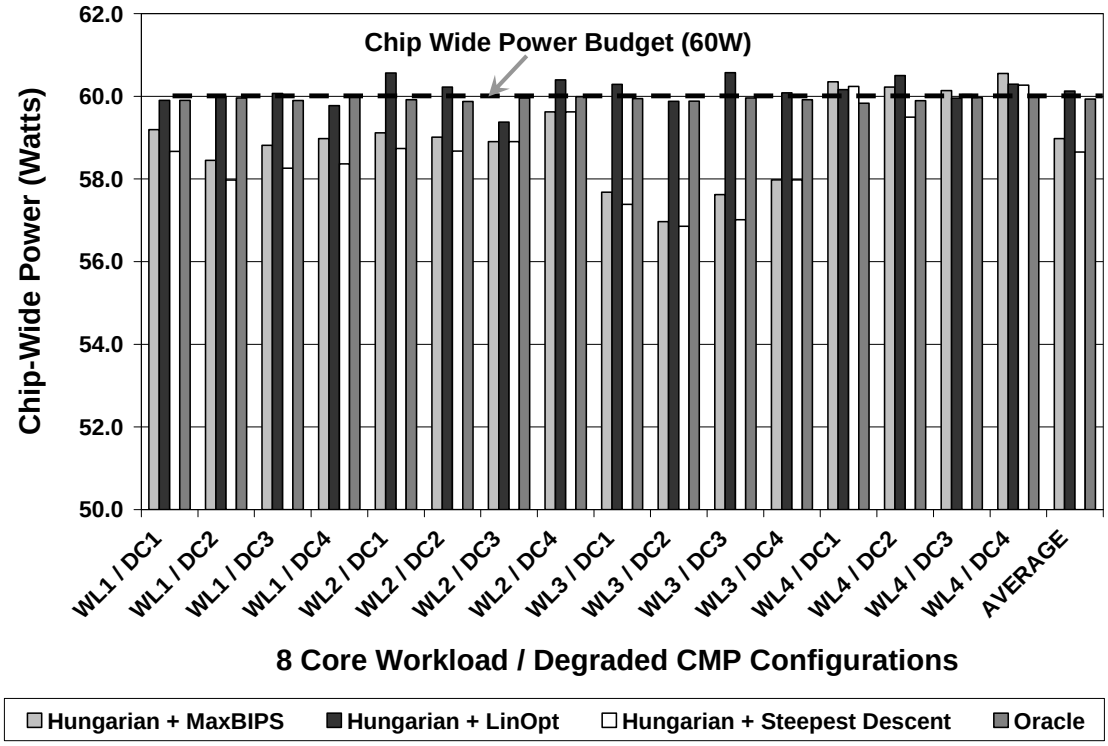


Figure 7.9: Power dissipation of uncoordinated scheduling and global power management algorithms and the oracle manager.

The performance losses for the three combinations versus the oracle are shown in Figure 7.8 for an eight core system with four different degraded configurations and four different workloads. The largest losses – at most 3% – occur when the independent algorithms slightly undershoot the power budget. These results prove our assertion that the two algorithms can operate independently while achieving near-optimal performance. Figure 7.9 shows the power dissipation results for these three

scheduling and power management algorithm combinations as well as the oracle. As expected the oracle manager never exceeds the budget. However, the MaxBIPS and Steepest Descent managers do exceed the budget on occasion due to their use of an imperfect voltage/performance/power model, although this occurs only on a couple of test cases. On the other hand, LinOpt, with its linear approximation of power using LLSF, consistently underestimates the power of its DVFS settings leading to budget overshoots on ten of the sixteen workload/degraded CMP combinations. Clearly this is an unacceptable situation and it will be addressed in Section 7.5.3 where we propose an improved version of LinOpt that uses a different method of linear approximation.

Using the oracle scheduler and power manager, we can gain some further insights into the impact of scheduling and power management in many-core processors. Since the oracle manager must examine all possible assignments and settings, we add extra functionality to the oracle to compute the average and worst case solutions while it is running. Figure 7.10 shows the percentage performance improvement of employing the optimal scheduling assignment, optimal power management settings, and both together in comparison to the average and worst cases of running the same managers. One observation is that finding a good scheduling assignment matters significantly more than the power setting. On reason for this is our assumption that in the average and worst case DVFS settings that the power manager still attempts to use up the full power budget. Consequently, these results show that it is most important for the power manager to find settings that use up the available power but less critical how that power is distributed. Regarding the results for applying both application scheduling and global power management, there is some additive affect when comparing the best manager over the average case, but less so when comparing worst cases. This occurs because the worst case when running scheduling and power management is typically the same scheduling assignment as the

worst case for scheduling alone. In summary, looking at the average results, with potential performance gains of over 15% and 37% over the average and worst case (respectively) for combined runtime managers, there is a clear motivation to pursue scalable and effective scheduling and power management algorithms for many-core architectures.

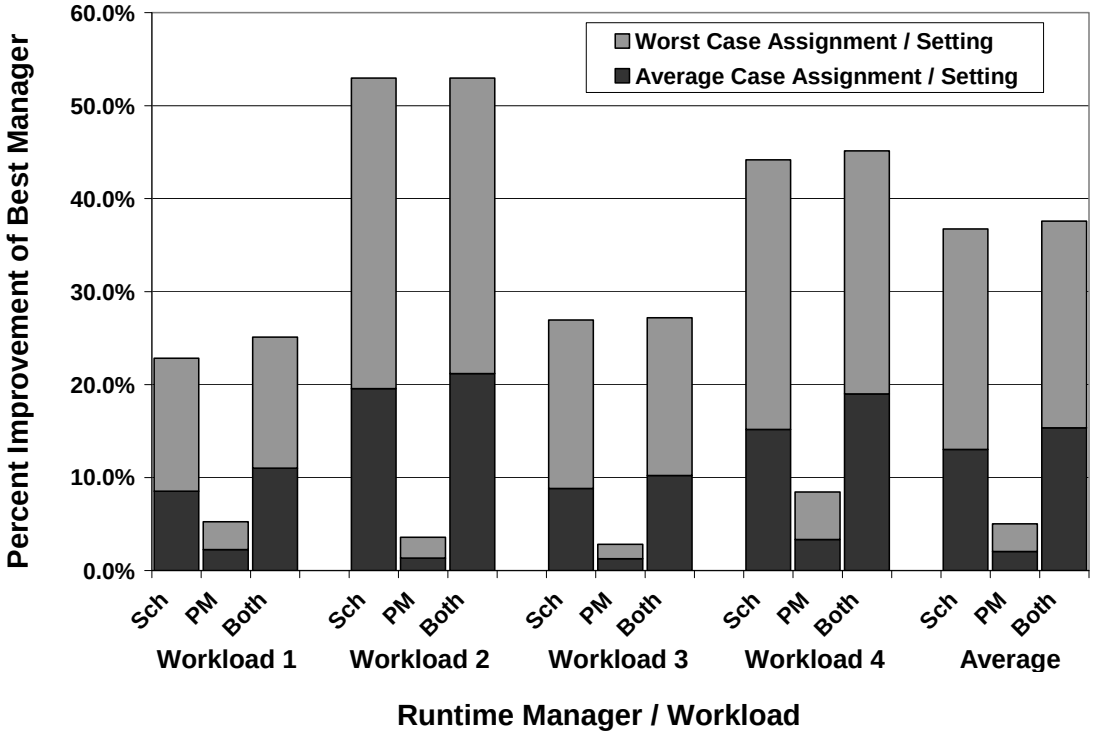


Figure 7.10: A study of the impact of scheduling and power management on many-core processors.

7.5.2. Application Scheduling Algorithms

We now evaluate the scheduling algorithms described in Section 7.3.2 in terms of their performance relative to the Hungarian SchedulingAlgorithm and their runtime overhead.

Figure 7.11 shows the runtime overhead of each algorithm over a range of four to 256 core organizations expressed as a percentage of the scheduling quantum. We implement both a sequential (SQ) version of the Hierarchical Hungarian Algorithm,

where a centralized scheduler makes the assignments for all the groups, and a parallel (PA) version where the scheduling task is partitioned among the groups of cores, leaving each group responsible for its own local schedule. For a small number of cores, the overhead for all scheduling algorithms is minor, but grows rapidly for the less-scalable Hungarian and Local Search algorithms, which have $O(n^3)$ and $O(n^2)$ complexity. Examining the results for the $O(n^4)$ Hungarian Algorithm, we immediately see that implementing the scheduler efficiently is very crucial, as this slower, more commonly seen version of the Hungarian Algorithm scales extremely poorly. On the other hand, the Greedy Scheduler and Sequential Hierarchical Hungarian Algorithm have low overheads even for 256 cores, but only the Parallel Hierarchical Hungarian Algorithm remains scalable beyond 256 cores.

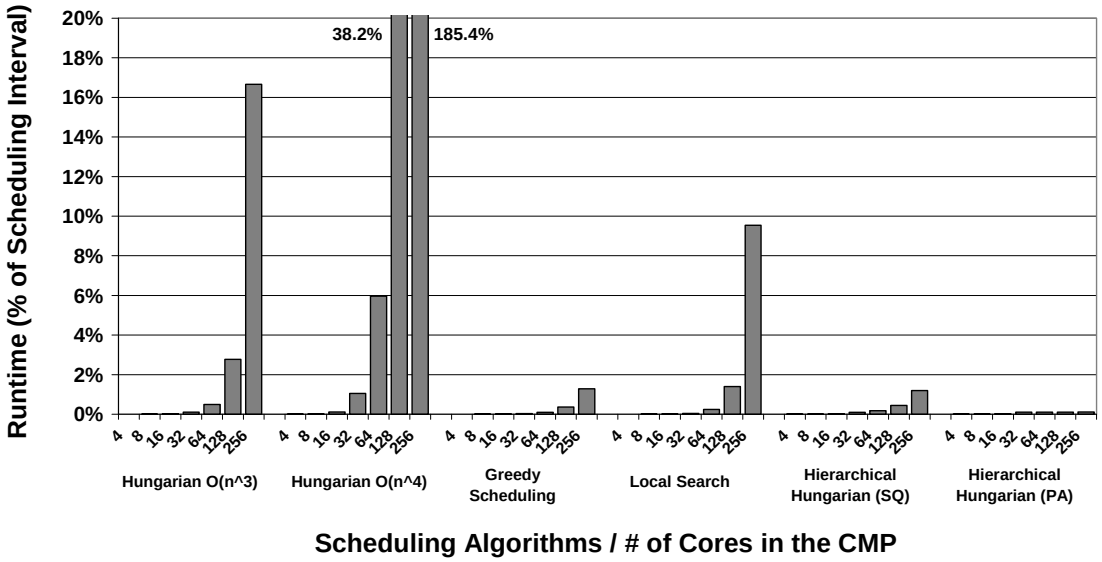


Figure 7.11: Scheduling algorithm runtimes as a percentage of the scheduling quantum.

While both the Sequential and Parallel Hierarchical Hungarian Algorithms are identically partitioned, the Sequential Algorithm is slowed down by the need to process sample results for the whole processor as well as construct the chip-wide scheduling assignment after all the Hungarian executions complete. Furthermore, the

Sequential Algorithm serializes the computation of the Hungarian Algorithm for each group in the hierarchy despite their complete independence. Together, the larger dataset and serialization slow down the sequential approach on a 256 core machine by an order of magnitude relative to the parallelized version. Even more impressive is the 150X speedup of the Parallel Hierarchical Hungarian Algorithm over the standard Hungarian Scheduler.

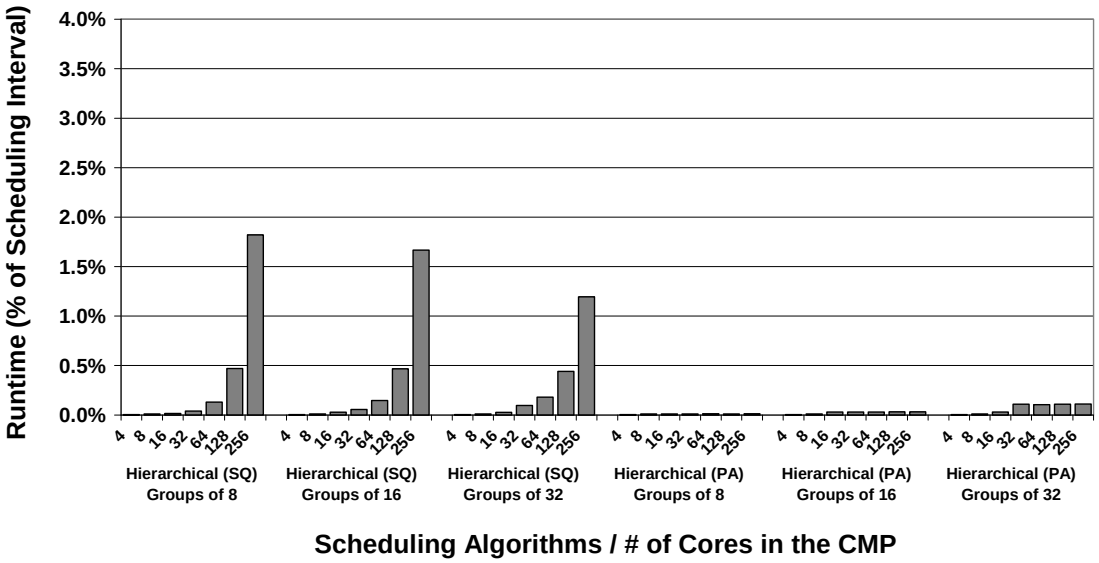


Figure 7.12: The impact of group size on runtime overhead for the Hierarchical Hungarian Algorithm.

The Hierarchical Hungarian Algorithm results presented in Figure 7.11 and Figure 7.14 are for groups of 32 cores. Figure 7.12 and Figure 7.13 present the runtime overhead and performance loss (respectively) of using groups of 8, 16, and 32 cores. We can see in Figure 7.12 that when implemented sequentially, it is preferable to uses larger group sizes. This is because with smaller group sizes, the Hungarian Algorithm must be executed serially more times and this cost outweighs the additional runtime for solving a larger assignment problem. For the parallel implementation, the overheads are very low for all cases. From Figure 7.13, we see that the performance loss relative to the Hungarian Algorithm doubles as the group size is halved. The

combination of the competitive runtime overhead and lower performance loss makes the use of groups of 32 cores the preferred choice for the Hierarchical Hungarian Algorithm in either sequential or parallel form.

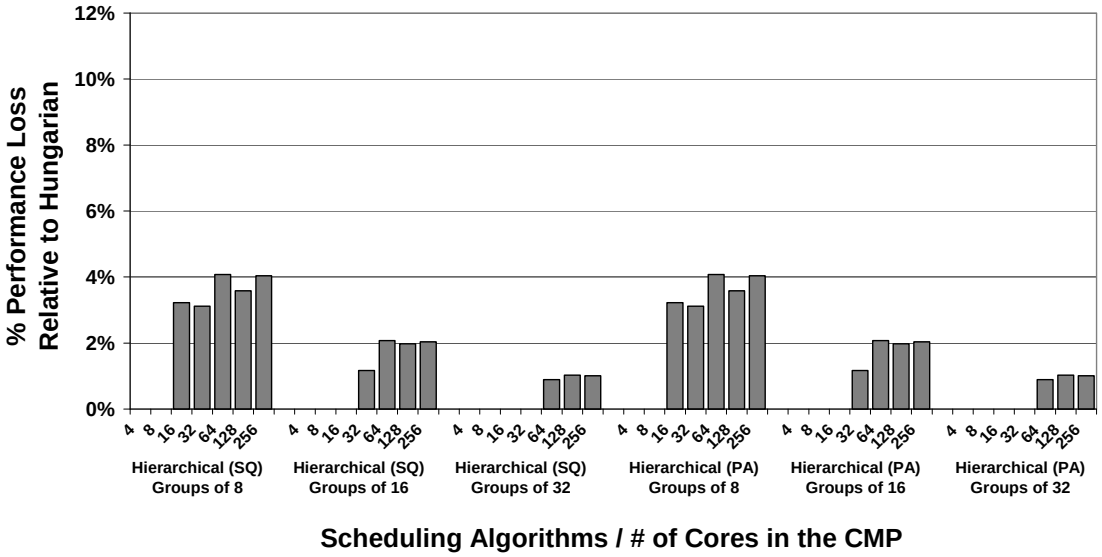


Figure 7.13: The impact of group size on performance loss for the Hierarchical Hungarian Algorithm.

Figure 7.14 shows the performance loss of the different algorithms relative to the performance of the Hungarian Algorithm. The $O(n^4)$ Hungarian Algorithm suffers no performance loss because it finds the same scheduling assignment. The Greedy Algorithm experiences the largest loss by far, 8-10% for the larger many-core organizations. This is not surprising considering that VarF&AppIPC's simple method of ranking application purely by IPC does not fully address the complexity of scheduling for unpredictably heterogeneous many-core architectures. A given application may suffer horribly due to the particular degradations on one core and have very high IPC on another. The average of this thread's IPC across these cores can be very misleading when trying to rank the thread as compute or memory bound. The performance of the Local Search Algorithm actually *improves* as the number of cores increases. This is due to Local Search's super-linear increase in search space

exploration (as discussed in Section 6.3.1) for the extra sampling intervals allocated for larger many-core chips. However, the improved performance is more than offset by the higher runtime overhead associated with the extra intervals. Both Hierarchical Hungarian Algorithms perform identically since they implement the same function, with an impressively small performance loss of about 1%. However, in considering both scalability and algorithm performance, the Parallel Hierarchical Hungarian Algorithm is the clear choice.

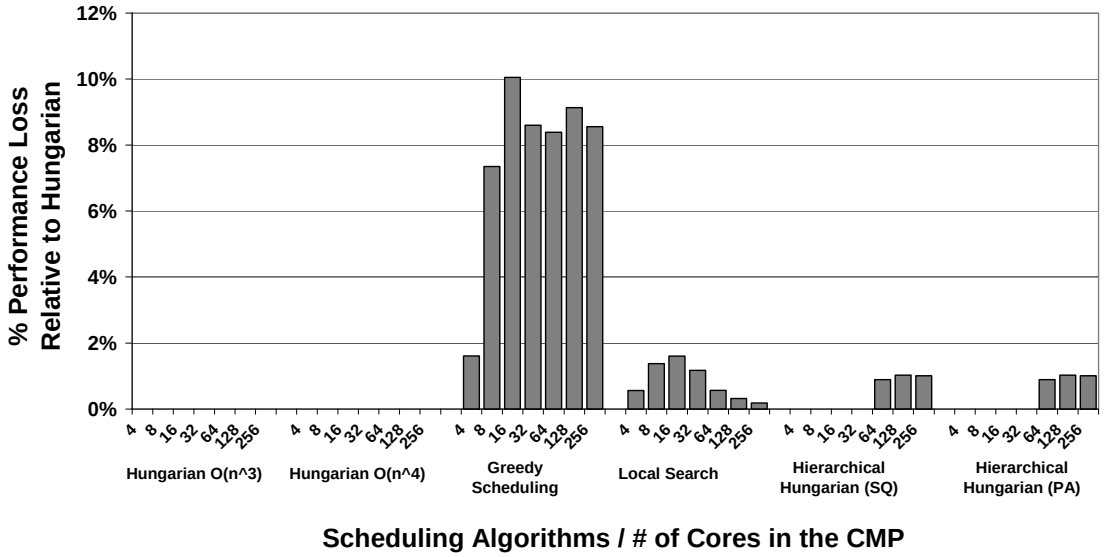


Figure 7.14: Scheduling algorithm performance percentage loss relative to the Hungarian Algorithm.

7.5.3. Global Power Management Algorithms

The runtime overhead and performance relative to LinOpt for the power management algorithms (from Section 7.3.3) are shown in Figure 7.15 and Figure 7.16, respectively. Due to its high-order polynomial average runtime, the overhead of LinOpt grows rapidly with the problem size (number of cores), and even exceeds the length of the power management quantum (10ms) for 256 cores. The LinOpt2 algorithm has slightly less runtime overhead compared to LinOpt because computing the equation of a line for linearly approximating power is simpler than using the linear

least squares fit method. However, the linear programming step dominates LinOpt2's runtime, making it scale just as poorly as LinOpt. Due to the fact that they have almost identical complexity (O(n·logn) versus O(p·n·logn)), the Greedy Algorithm and Steepest Descent Algorithm have about the same overheads, less than 2% for 256 cores, and provide 75X and 62X speedups (respectively) over LinOpt.

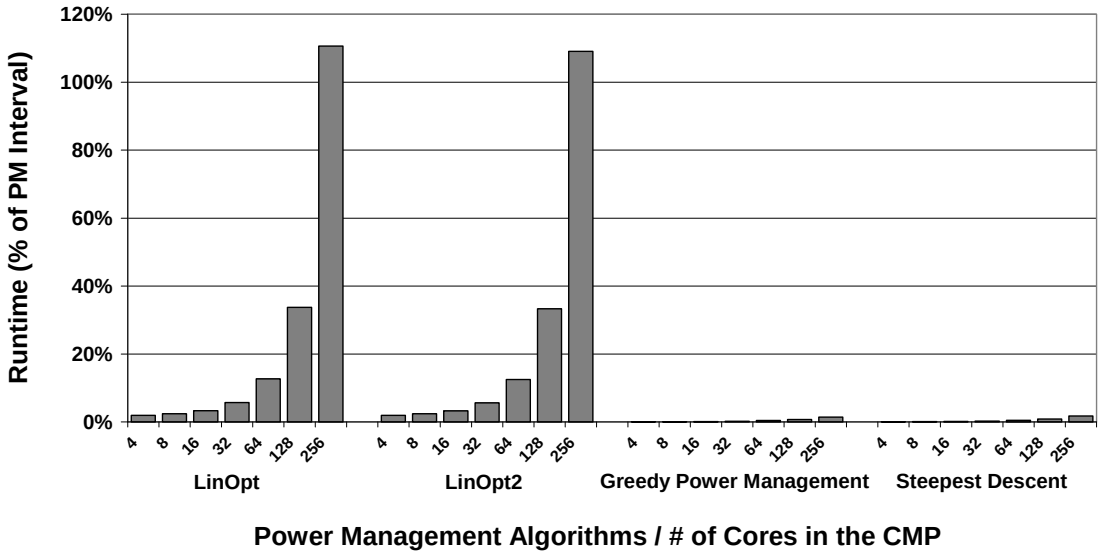


Figure 7.15: Global power management algorithm runtimes as a percentage of the power management quantum.

Looking at the performance results shown in Figure 7.16, Steepest Descent slightly outperforms Greedy and even slightly outperforms LinOpt in all cases. It should be noted that the LinOpt power settings actually cause the processor to exceed the chip-wide power budget in almost all test cases. This occurs because of the underestimation of the power dissipation caused by the LLSF method as mentioned in Section 7.3.3. Consequently, LinOpt's performance is actually unfairly better than would occur if it used the appropriate amount of power. LinOpt2's power dissipation does not exceed the power budget and thus its performance estimates are a more accurate representation of what can be accomplished by linear optimization.

Relative to LinOpt2, Steepest Descent has about a one percent performance advantage. This is due to the inaccuracy in the linear voltage/power relationship which causes LinOpt and LinOpt2 to sometimes favor assigning power to the wrong applications. On the other hand, Steepest Descent is calibrated to use an aggressive voltage/power relationship of $P \propto V^{2.5}$ which allows the algorithm to very tightly match the global power budget and extract as much performance as possible. Due to the extremely low runtime overhead of Steepest Descent and its very competitive performance numbers, we chose not to pursue hierarchical GPM techniques.

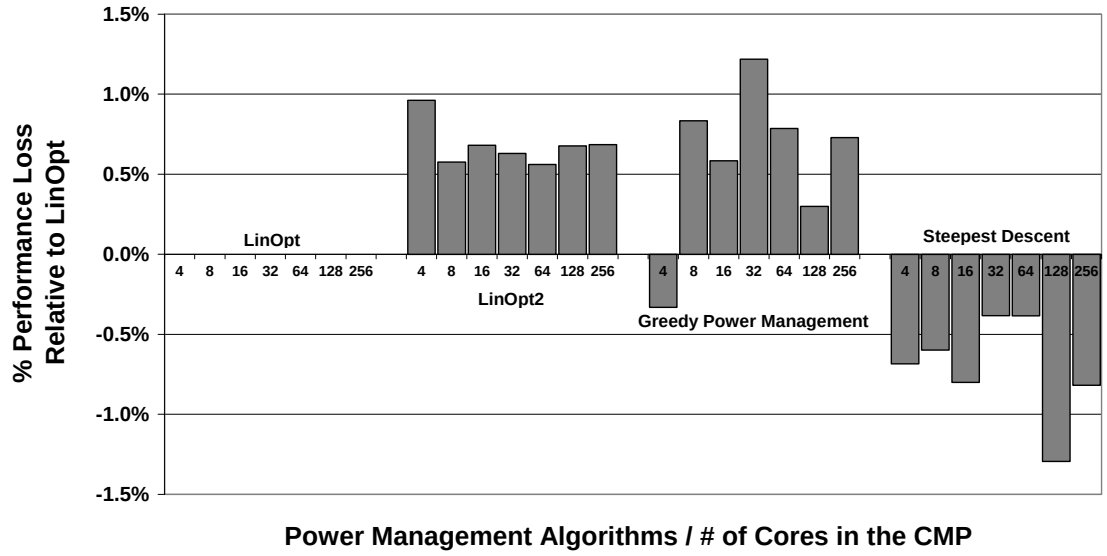


Figure 7.16: Power management algorithm performance percentage loss relative to the LinOpt algorithm.

7.5.4. Online Results

As discussed in Section 7.4.1, we simulate full length runs to capture the affects of online experiments which include application phase changes and dynamic behavior. This study also tests whether our scheduling and power management algorithms are effective with realistic lengths for sampling intervals. In this study, we employ oracle versions of the Hungarian Scheduling Algorithm and LinOpt which

ignore the computational overhead of the algorithms in order to focus the study on algorithm performance and sampling accuracy.

Figure 7.17 shows one example of our online performance results from simulations with 256 core CMPs and two different workloads and two degraded configurations. We run these simulations for two application scheduling quanta, for a total simulation time of two weeks on a 512 core server cluster. The results confirm the conclusions of Figure 7.14 and Figure 7.16. In particular, we demonstrate that the Hierarchical Hungarian Algorithm achieves similar performance to the Hungarian Algorithm. In fact, due to the limited amount of time available for sampling in the online environment, the Hungarian Algorithm is unable to take as accurate samples as it does in the offline study, and thus the Hierarchical Algorithm does somewhat better than in the offline study. The superior performance of the Hierarchical Hungarian Algorithm is consistent across both LinOpt and the Steepest Descent Algorithm.

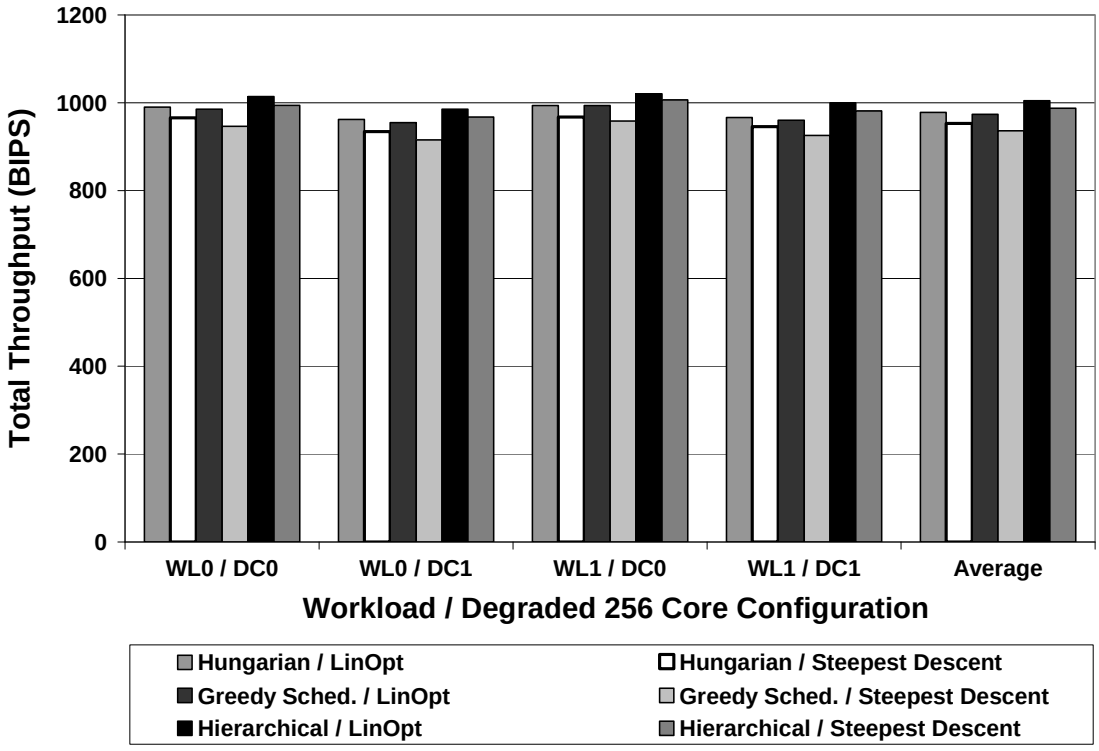


Figure 7.17: Online performance results for application scheduling and power management algorithms with 256 core CMPs over 2 application scheduling quanta.

We simulated the Hierarchical Hungarian Scheduler both with (shown in Figure 7.17) and without randomized application groupings and found that for these workloads, randomization improved the results by a small but consistent amount. However, we would expect bigger gains when running more scheduling quanta because our two quanta simulations did not give the algorithm enough time to converge.

Regarding global power management, Steepest Descent performs nearly as well LinOpt. These results are encouraging because as in Section 7.5.3, LinOpt's results are unrealistic because the model is underestimating the power dissipation across all workloads and degraded configurations leading to global budget overshoots during most of the runtime. In summary, the results of these full length simulations indicate that our scalable Hierarchical Hungarian Scheduling Algorithm and Steepest Descent power management algorithm will provide equal or better performance compared to oracle versions of the best algorithms proposed in prior work.

7.6. Conclusions

Future many-core microprocessors containing hundreds of cores will need to tolerate manufacturing defects, wear-out failures, and extreme process variations. The resulting dynamic heterogeneity of these systems requires intelligent, yet highly scalable, runtime scheduling and power management algorithms. In this chapter, we perform a detailed analysis of the effectiveness and scalability of a range of algorithms for many-core systems of up to 256 cores. First, we show that there is no need to coordinate scheduling and global power management, which greatly reduces the search space for runtime power-performance optimization. We develop the Hierarchical Hungarian Algorithm for application scheduling and demonstrate that it is up to 150X faster than the Hungarian Algorithm while providing a mere 1% less

maximum throughput. We also demonstrate that the Steepest Descent power management algorithm has 75X less runtime overhead (for 256 cores) than the state-of-the-art LinOpt algorithm, does not have the problem of regularly exceeding the chip-wide budget, and even provides some performance gains. Finally, we validate these findings by running extensive online simulations which confirm both the scalability and performance capability of our proposed scheduling and power management algorithms. These scalable techniques will be essential in future many-core systems in order to mitigate the deleterious effects of variations and hardware degradation.

CHAPTER 8

OPPORTUNITIES FOR FUTURE WORK

There are a number of directions in which my dissertation research can be expanded, including further investigating the TAPAS algorithm from Chapter 5, combining our schedulers and power managers with Complexity-Adaptive Processing, exploring runtime management of parallel applications, improving our simulation infrastructure, and studying degradation- and phase-aware algorithms for runtime management.

One clear extension would be to incorporate the TAPAS algorithm into the scalability study of Chapter 7 to assess how this scheme compares to more contemporary GPM approaches on future many-core processors. The power allocation concept is a very general paradigm for power management where the global layer focuses directly on dividing the power budget among the cores and the individual cores manage their own voltage and frequency settings. TAPAS could be implemented using linear programming or Steepest Descent, much as Teodorescu and Torrellas [134] develop LinOpt and Chapter 7 describes Steepest Descent as alternatives to using the brute force MaxBIPS algorithm for the global power manager. Thus, we would expect these other implementations to provide dramatic scalability improvements over the brute force implementation of TAPAS in Chapter 5 but experimental results are needed to validate this theory. In addition, by focusing on power directly, TAPAS provides increased robustness to error in the power-performance model. Futhermore, TAPAS use of fine-grained per-core DVFS control should enable this algorithm to reduce power budget violations. The scalability of these attributes could be assessed for future many-core architectures.

Another future direction would be to explore Complexity-Adaptive Processing [3][4][5][98] within the context of future unreliable multi-core chips. These techniques

improve energy efficiency by selectively disabling unneeded portions of the chip's processing resources in order to save power with low performance overhead. Adding adaptivity to the microarchitecture provides another dimension, in addition to thread scheduling and power management, by which to mitigate the effects of process variations and hard errors, maintain power-performance efficiency, and maximize throughput under a global power budget. With three different techniques at its disposal, the adaptive thread manager would have a tremendous number of possible configurations to explore, likely requiring the investigation of new highly scalable algorithms.

Chapters 4 through 7 focus on workloads consisting of multiprogrammed sequential applications and try to leverage application diversity by matching it to the dynamic heterogeneity of the CMP. This study could be expanded to included parallel applications which are predicted to become widespread because they best harness the potential of future many-core processors. Multithreaded applications are more challenging workloads in one respect, because inter-core interactions become crucial and the impact of the chip interconnect is critical. However, the threads in most parallel applications perform rather homogeneous tasks leading to uniform execution behavior. This feature may actually make it easier for scheduling and power management algorithms to provide high performance and power efficiency. One problem that the adaptive thread manager might have to address is that the most degraded core assigned to a thread in the parallel application will likely become the bottleneck slowing down the whole application. On the other hand, the runtime optimization problem may shift to managing multiple multithreaded applications running simultaneously on the many-core processor rather than addressing the individual threads of a single parallel application. Workloads consisting of a mix of sequential and parallel applications could also be considered.

The performance of a parallel application is highly dependent on the interactions between the cores, including the impact of the cache hierarchy and interconnect. In order to best investigate these workloads, our simulation infrastructure would need to be expanded to model cache sharing effects, on-chip interconnection networks, and off-chip bandwidth in more detail. At the same time, the impact of process variations and hard errors on die components outside the core could be studied and methods to mitigate these effects could be designed.

Thus far, the application scheduling and global power management techniques that we have examined operate by dynamically sampling the applications on the different cores in the CMP and trying various power settings to learn how the threads behave. The algorithms then assign threads to cores and apply DVFS based on these findings. However, another approach is possible. Since processor wear-out occurs at much longer time frames than OS scheduling intervals, it is possible for the adaptive thread manager to maintain a knowledge base of core degradation attributes to make informed scheduling decisions without sampling. Likewise, application phase behavior can be studied and predicted [121] allowing the threads to be assigned to cores that match their level of compute- and memory-boundedness. The global power manager could also use knowledge of core capabilities and application needs to maximize throughput without having to rely on possibly inaccurate sampling information. This methodology would allow the adaptive thread manager to circumvent the problem of collecting the large numbers of samples required for future many-core architectures. However, incorporating degradation-awareness and phase analysis will add significant complexity to the chip multiprocessor. Our techniques have the advantage of a relatively simple implementation that relies only on direct sampling of performance and power and does not need a deep understanding of application/core interactions. This simplicity also provided robustness in the presence

of variability and faults. Future work could explore the many tradeoffs of these two adaptive thread management philosophies.

CHAPTER 9

CONCLUSIONS

Future processor architectures will require intelligent adaptive thread management policies to prevent power dissipation, thermal control, and reliability problems from halting the trend of continued performance increases in every technology generation. In this dissertation, we investigate two crucial adaptive thread management techniques, thread migration and power management, and develop novel schemes for their implementation in clustered SMT architectures and heterogeneous chip multiprocessors.

In clustered SMT architectures, we apply these techniques within a processor core to provide dynamic thermal management. We employ the back-end steering mechanism inherent in a clustered SMT to provide low cost DTM which leverages spatial non-uniformity in temperature due to variations in application behavior to cool hotspots on the die. We also combine our DTM steering mechanism with dynamic voltage and frequency scaling to provide robust thermal control across all workload types.

Process variations, manufacturing defects, and wear-out errors will create dynamic heterogeneity among the cores of future chip multiprocessors. We mitigate the impact of these reliability issues by developing OS scheduling algorithms, based on the Hungarian Algorithm and Local Search that match application needs to available resources of the degraded cores. We also devise a new paradigm for global power management, called TAPAS, which directly controls power, rather than using voltage and frequency control as a proxy, delivering increased performance, accuracy, and tolerance to variability.

Finally, we present the first comprehensive and theoretically rigorous analysis of the scalability of application scheduling and global power management algorithms

for unpredictably heterogeneous many-core processors. Studying architectures with four to two-hundred and fifty-six cores, we find that the high computational overhead and sampling requirements of contemporary runtime managers designed for small-scale CMPs will prevent their use as the number of cores on a chip increases. We address this deficiency by creating a Hierarchical Hungarian Algorithm for application scheduling and adapting Steepest Descent for global power management and show that both are highly scalable due to their high performance and low computational complexity across chip multiprocessors of all sizes.

REFERENCES

[1] A. Aggarwal, B.C. Paul, H. Mahmoodi, A. Datta, and K. Roy. A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Jan. 2005, 13(1):27-38.

[2] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors. In the *34th International Symposium on Computer Architecture (ISCA)*, June 2007, pp. 470-481.

[3] D.H. Albonesi. Dynamic IPC/Clock Rate Optimization. In the *25th International Symposium on Computer Architecture (ISCA)*, June 1998, pp. 282-292.

[4] D.H. Albonesi. The Inherent Energy Efficiency of Complexity-Adaptive Processors. In the *Workshop on Power-Driven Microarchitecture*, June 1998.

[5] D.H. Albonesi, R. Balasubramonian, S.G. Dropsho, S. Dwarkadas, E.G. Friedman, M.C. Huang, V. Kursun, G. Magklis, M.L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P.W. Cook, and S.E. Schuster. Dynamically Tuning Processor Resources with Adaptive Processing. In *IEEE Computer*, Dec. 2003, 36(12):49-58.

[6] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In the *32nd International Symposium on Microarchitecture (MICRO)*, Nov. 1999, pp. 196-207.

[7] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In the *32nd International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 506-517.

[8] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In the *30th International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 275-286.

[9] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *ACM International Conference on Computing Frontiers (CF)*, 2006, pp. 29-39.

[10] R. Bergamaschi, G. Han, A. Buyuktosunoglu, H. Patel, I. Nair, G. Dittmann, G. Janssen, N. Dhanwada, Z. Hu, P. Bose, and J. Darringer. Exploring Power Management in Multi-Core Systems. In the *13th Asian and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2008, pp. 708-713.

[11] R. Bitirgen, E. Ipek, and J.F. Martinez, Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach, In the *41st International Symposium on Microarchitecture (MICRO)*, Nov. 2008, pp. 318-329.

[12] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-Calibrating Online Wearout Detection. In the *40th International Symposium on Microarchitecture (MICRO)*, Dec. 2007, pp. 109-122.

[13] S. Borkar, Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation, *IEEE Micro*, Nov./Dec. 2005, 25(6):10-16.

[14] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter Variations and Impact on Circuits and Microarchitecture. *Design Automation Conference (DAC)*, June 2003, 21.1:338-342.

[15] F.A. Bower, S. Olev, and D.J. Sorin. Autonomic Microprocessor Execution via Self-Repairing Arrays. In *IEEE Transactions on Dependable and Secure Computing*, Oct.-Dec. 2005, 2(4):297-310.

[16] F.A. Bower, D. J. Sorin, and L.P. Cox. The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling. In *IEEE Micro*, May/June 2008, 28(3):17-25.

[17] F.A. Bower, P.G. Shealy, S. Orev, and D.J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In the *34th International Conference on Dependable Systems and Networks (DSN)*, June 2004, pp. 51-60.

[18] F.A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In the *38th International Symposium on Microarchitecture (MICRO)*, Nov. 2005, pp. 197-208.

[19] F.A. Bower, D. J. Sorin, and S. Ozev. Online Diagnosis of Hard Faults in Microprocessors. In *ACM Transactions on Architecture and Code Optimization (TACO)*, June 2007, 4(2):8.1-8.32.

[20] K.A. Bowman, A.R. Alameldeen, S.T. Srinivasan, and C.B. Wilkerson. Impact of Die-to-Die and Within-Die Parameter Variations on the Throughput Distribution of Multi-Core Processors. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2007, pp. 50-55.

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

[22] D. Brooks and M. Martonosi Dynamic Thermal Management for High-Performance Microprocessors. In the *7th International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 171-182.

[23] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In the *27th International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 83-94.

[24] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. In *ACM SIGARCH Computer Architecture News*, June 1997. 23(3):13-25.

[25] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Published by the Society of Industrial and Applied Mathematics, Philadelphia, PA, 2009, pp. 73-87.

[26] J.A. Butts and G.S. Sohi. A Static Power Model for Architects. In the 33rd *International Symposium on Microarchitecture (MICRO)*, Dec. 2000, pp. 191-201.

[27] J.M. Cebrián, J.L. Aragón, J.M. Garcia, P. Petoumenos, and S. Kaxiras. Efficient Microarchitecture Policies for Accurately Adapting to Power Constraints. In the *24th International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.

[28] P. Chaparro, J. González, and A. González. Thermal-Aware Clustered Microarchitectures. In the *International Conference on Computer Design (ICCD)*, Oct. 2004, pp. 48-53.

[29] P. Chaparro, J. González, and A. González. Thermal-Effective Clustered Microarchitectures. In the *1st Workshop on Temperature-Aware Computer Systems (TACS)*, June 2004.

[30] P. Chaparro, J. González, G. Magklis, Q. Cai, and A. González. Understanding the Thermal Implications of Multicore Architectures. In *IEEE Transactions On Parallel and Distributed Systems (TPDS)*, Aug. 2007, 18(8):1055-1065.

[31] P. Chaparro, G. Magklis, J. González, and A. González. Distributing the Frontend for Temperature Reduction. In the *11th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2005, pp. 61-70.

[32] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. In the *33rd International Symposium on Microarchitecture (MICRO)*, Dec. 2000, pp. 87-97.

[33] P. Chen, C.-C. Chen, C.-C. Tsai, and W.-F. Lu. A Time-to-Digital-Converter-Based CMOS Smart Temperature Sensor. In *IEEE Journal of Solid-State Circuits*, Aug. 2005, 40(8):1642-1648.

[34] J.D. Collins and D.M. Tullsen. Clustered Multithreaded Architectures – Pursuing Both IPC and Cycle Time. In the *18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004, pp. 76-86.

[35] A. Das, S. Ozdemir, G. Memik, J. Zambreno, and A. Choudhary. Microarchitectures for Managing Chip Revenues under Process Variations. In *IEEE Computer Architecture Letters*, June 2007, 6(2):29-32.

[36] A. Das, S. Ozdemir, G. Memik, J. Zambreno, and A. Choudhary. Mitigating the Effects of Process Variations: Architectural Approaches for Improving Batch Performance. In the *Workshop on Architectural Support for Gigascale Integration (ASGI)*, June 2007.

[37] M. DeVuyst, R. Kumar, and D.M. Tullsen. Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In the *20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[38] A. Dhodapkar, C.H. Lim, G. Cai, and W.R. Daasch. TEM^{2}P^{2}EST: A Thermal Enabled Multi-Model Power/Performance ESTimator. In the *1st Workshop on Power-Aware Computer Systems (PACS)*, Nov. 2000.

[39] J. Donald and M. Martonosi. Temperature-Aware Design Issues for SMT and CMP Architectures. In the *5th Workshop on Complexity-Effective Design (WCED)*, June 2004.

[40] J. Donald and M. Martonosi. Leveraging Simultaneous Multithreading for Adaptive Thermal Control. In the *2nd Workshop on Temperature-Aware Computer Systems (TACS)*, June 2005.

[41] J. Donald and M. Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. In the *33rd International Symposium on Computer Architecture (ISCA)*, June 2006, pp. 78-88.

[42] J. Donald and M. Martonosi. Power Efficiency for Variation-Tolerant Multicore Processors. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Oct. 2006, pp. 304-309.

[43] M. Ekman and P. Stenstrom. Performance and Power Impact of Issue-Width in Chip-Multiprocessor Cores. In the *International Conference on Parallel Processing (ICPP)*. Oct. 2003, pp. 359-368.

[44] A. El-Moursy, R. Garg, D.H. Albonesi, and S. Dwarkadas. Partitioning Multi-Threaded Processors with a Large Number of Threads. In the *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2005, pp. 112-123.

[45] A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling on Heterogeneous Core Systems. In the *Workshop on Operating System Support for Heterogeneous Multicore Architectures (OSHMA)*, Sept. 2007.

[46] S. Feng, S. Gupta, and S. Mahlke. Olay: Combat the Signs of Aging with Introspective Reliability Management. In the *Workshop in Quality-Aware Design (W-QUAD)*, June 2008.

[47] S. Ghiasi and D. Grunwald. Design Choices for Thermal Control in Dual-Core Processors. In the *5th Workshop on Complexity-Effective Design (WCED)*, June 2004.

[48] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for Heterogeneous Processors in Server Systems. *ACM International Conference on Computing Frontiers (CF)*, May 2005, pp. 199-210.

[49] I. Griva, S.G. Nash, and A. Sofer. Linear and Nonlinear Optimization. Published by the Society of Industrial and Applied Mathematics, Philadelphia, PA, 2009, pp.301-317.

[50] J. Hasan, A. Jalote, T.N. Vijaykumar, and C.E. Brodley. Heat Stroke: Power-Density-Based Denial of Service in SMT. In the *11th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2005, pp. 166-177.

[51] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density Through Activity Migration. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2003, pp. 217-222.

[52] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2007, pp. 38-43.

[53] S. Herbert and D. Marculescu. Variation-Aware Dynamic Voltage/Frequency Scaling, In the *15th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2009, pp. 301-312.

[54] M. Huang, J. Renua, S. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In the *33rd International Symposium on Microarchitecture (MICRO)*, Dec. 2000, pp. 202-213.

[55] E. Humenay, D. Tarjan, and K. Skadron. Impact of Parameter Variations on Multi-Core Chips. In the *Workshop on Architectural Support for Gigascale Integration (ASGI)*, June 2006.

[56] E. Humenay, D. Tarjan, and K. Skadron. Impact of Process Variations on Multi-Core Performance Symmetry. In *Design, Automation and Test in Europe (DATE)*, April 2007, pp. 1653-1658.

[57] Intel Corporation, From a Few Cores to Many: A Tera-scale Computing Research Overview, Whitepaper, 2006

[58] C. Isci, A. Buyuktosunoglu, C-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In the *39th International Symposium on Microarchitecture (MICRO)*, Dec. 2006, pp. 347-358.

[59] P. Ituero, J.L. Ayala, M. López-Vallejo. Leakage-Based On-Chip Thermal Sensor for CMOS Technology. In the *International Symposium on Circuits and Systems (ISCAS)*, May 2007, pp. 3327-3330.

[60] A.Z. Jooya, A. Baniasadi, and M. Analoui. History-Aware, Resource-Based Dynamic Scheduling for Heterogeneous Multi-core Processors. In the *3rd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, June 2009.

[61] R. Joseph. Exploring Salvage Techniques for Multi-core Architectures. In the *2nd Workshop on High Performance Computing Reliability Issues (HPCRI)*, Feb 2006.

[62] P. Juang, Q. Wu, L-S. Peh, M. Martonosi, and D. W. Clark. Coordinated, Distributed, Formal Energy Management of CMP Multiprocessors. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2005, pp. 127-130.

[63] K. Kang, K. Kim, A.E. Islam, M.A. Alam, and K. Roy. Characterization and Estimation of Circuit Reliability Degradation under NBTI using On-Line I_{DDQ} Measurement. In the *44th Design Automation Conference (DAC)*, June 2007, 20.1:358-363.

[64] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In the *14th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2008, pp. 123-134.

[65] A. Kumar, L. Shang, L.-S. Peh, and N.K. Jha. HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management. In the *43rd Design Automation Conference (DAC)*, July 2006, 33.3:548-553.

[66] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In the *36th International Symposium on Microarchitecture (MICRO)*, Dec. 2003, pp. 81-92.

[67] R. Kumar, D.M. Tullsen, and N.P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In the *15th International Symposium on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2006, pp. 23-32.

[68] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In the *34th International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 64-75.

[69] C. LaFrieda, E. İpek, J.F. Martínez, and R. Manohar. Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In the *37th International Conference on Dependable Systems and Networks (DSN)*, June 2007, pp. 317-326.

[70] F. Latorre, J. González, and A. González. Back-End Assignment Schemes for Clustered Multithreaded Processors. In the *18th International Conference on Supercomputing (ICS),* June 2004, pp. 316-325.

[71] B. Lee and D. Brooks. Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency. In the 6th *Workshop on Complexity-Effective Design (WCED)*, June 2005.

[72] J. Li and J.F. Martínez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In the *12th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 77-87.

[73] T. Li, D. Baumberger, D.A. Koufaty, and S. Hahn, Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures, *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2007.

[74] Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In the *11th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2005, pp. 71-82.

[75] Y. Li, D. Brooks, Z. Hu, K. Skadron, and P. Bose. Understanding the Energy Efficiency of Simultaneous Multithreading. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004, pp. 44-49.

[76] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP Design Space Exploration Subject to Physical Constraints. In the *12th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 17-28.

[77] X. Liang and D. Brooks. Microarchitecture Parameter Selection to Optimize System Performance Under Process Variation. *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2006, pp. 429-436.

[78] X. Liang and D. Brooks. Mitigating the Impact of Process Variations on Processor Register Files and Execution Units. In the *39th International Symposium on Microarchitecture (MICRO)*, Dec. 2006, pp. 504-514.

[79] X. Liang, G.-Y. Wei, and D. Brooks. ReVIVaL: A Variation-Tolerant Architecture Using Voltage Interpolation and Variable Latency. In the *35th International Symposium on Computer Architecture (ISCA)*, June 2008, pp. 191-202.

[80] W. Liao, L. He, and K.M. Lepak. Temperature and Supply Voltage Aware Performance and Power Modeling at Microarchitecture Level. In *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, July 2005, 24(7):1042-1053.

[81] W. Liao, F. Li, and L. He. Microarchitecture Level Power and Thermal Simulation Considering Temperature Dependent Leakage Model. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2003, pp. 211-216.

[82] C.H. Lim, W.R. Daasch, and G. Cai. A Thermal-Aware Superscalar Microprocessor. In the *3rd International Symposium on Quality Electronic Design (ISQED)*, March 2002, pp. 517-522.

[83] A.J. Martin. Towards an Energy Complexity of Computation. *Information Processing Letters*. Feb. 2001, 77(2-4):181-187.

[84] R. McGowen, C.A. Poirier, C. Bostak, J. Ignowski, M. Millican, W.H. Parks, and S. Naffziger. Power and Temperature Control on a 90-nm Itanium Family Processor. In the *Journal of Solid-State Circuits*, Jan. 2006, 41(1):229-237.

[85] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-cost, Comprehensive Error Detection in Simple Cores. In the *40th International Symposium on Microarchitecture (MICRO)*, Dec. 2007, pp. 210-222.

[86] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-cost, Comprehensive Error Detection in Simple Cores. In *IEEE Micro*, Jan./Feb. 2008, 28(1):52-59.

[87] A. Meixner and D.J. Sorin. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In the *38th International Conference on Dependable Systems and Networks (DSN)*, June 2008, pp. 80-89.

[88] K. Meng, R. Joseph, R.P. Dick, and L. Shang, Multi-Optimization Power Management for Chip Multiprocessors, In *17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2008, pp. 177-186.

[89] A. Merkel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. In *EuroSys*, April 2006, pp. 403-414.

[90] L. Miao, Y. Qi, D. Hou, and Y. Dai. Energy-Aware Scheduling Tasks on Chip Multiprocessor. In the *3rd International Conference on Natural Computation (ICNC)*, Aug. 2007, Volume 4, pp. 319-323.

[91] M. Monchiero, R. Canal, and A. González. Design Space Exploration for Multicore Architectures: A Power/Performance/Thermal View. In the *20th International Conference on Supercomputing (ICS),* June 2006, pp. 177-186.

[92] T. Mudge. Power: A First-Class Architectural Design Constraint. In *IEEE Computer*. April 2001, 34(4):52-58.

[93] J. Munkres. Algorithms for Assignment and Transportation Problems. In the *Journal of the Society of Industrial and Applied Mathematics*, March 1957, 5(1):32-38.

[94] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou. Yield-Aware Cache Architectures. In the *39th International Symposium on Microarchitecture (MICRO)*, Dec. 2006, pp. 15-25.

[95] F. Paterna, L. Benini, A. Acquaviva, F. Papariello, G. Desoli, and M. Olivieri. Adaptive Idleness Distribution for Non-Uniform Aging Tolerance in MultiProcessor Systems-on-Chip. In *Design, Automation and Test in Europe (DATE)*, April 2009, pp. 906-909.

[96] M. Pertijs, K. Makinwa, and J. Huijsing. A CMOS Temperature Sensor with a 3σ Inaccuracy of ±0.1°C from -55°C to 125°C. In the *International Solid-State Circuits Conference (ISSCC)*, Feb. 2005, 13.1:238-239,596.

[97] R.A. Pilgrim. *Munkres' Assignment Algorithm.* http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html, 2008.

[98] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirement of Instruction Scheduling through Dynamic Allocation of Multiple Datapath Resources. In the *34th International Symposium on Microarchitecture (MICRO)*, Dec. 2001, pp. 90-101.

[99] M.D. Powell, E. Schuschman, and T.N. Vijaykumar. Balancing Resource Utilization to Mitigate Power Density in Processor Pipelines. In the *38th International Symposium on Microarchitecture (MICRO)*, Nov. 2005, pp. 294-304.

[100] M.D. Powell, M. Gomaa, and T.N. Vijaykumar. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In the *XI International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004, pp. 260-270.

[101] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing, 2nd Edition*. Published by the Cambridge University Press, Cambridge, England, 1992, pp. 656-666.

[102] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In the *12th International Symposium on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2003, pp. 15-26.

[103] J.M. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Published by Prentice-Hall, Englewood Cliffs, NJ, 2003.

[104] P. Ramachandran, S.V. Adve, P. Bose, and J.A. Rivers. Metrics for Architectural-Level Lifetime Reliability Analysis. In the *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008, pp. 202-212.

[105] K.K. Rangan, G.-Y. Wei, and D. Brooks. Thread Motion: Fine-Grained Power Management for MultiCore Systems, In the *36th International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 302-313.

[106] C.R. Reeves (Editor). *Modern Heuristic Techniques for Combinatorial Problems*. Published by McGraw-Hill Book Company, London, UK, 1995.

[107] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. *SESC Simulator*. http://sesc.sourceforge.net, 2005.

[108] D. Roberts, R.G. Dreslinski, E. Karl, T. Mudge, D. Sylvester, and D. Blaaux. When Homogeneous becomes Heterogeneous. In the *Workshop on Operating System Support for Heterogeneous Multicore Architectures (OSHMA)*, Sept. 2007.

[109] B.F. Romanescu, M.E. Bauer, S. Olev, and D.J. Sorin. Reducing the Impact of Intra-Core Process Variability with Criticality-Based Resource Allocation and Prefetching. In the *ACM International Conference on Computing Frontiers (CF)*, May 2008, pp. 129-138.

[110] B.F. Romanescu, M.E. Bauer, S. Olev, and D.J. Sorin. VariaSim: Simulating Circuits and Systems in the Presence of Process Variability. In *ACM SIGARCH Computer Architecture News*, Dec. 2007, 35(5):45-48.

[111] B.F. Romanescu, S. Olev, and D.J. Sorin. Quantifying the Impact of Process Variability on Microprocessor Behavior. In the *2nd Workshop on Architectural Reliability (WAR)*, Dec. 2006.

[112] S.M. Sait and H. Youssef. *Iterative Computer Algorithms with Applications in Engineering*. Published by the IEEE Computer Society, Los Alamitos, CA, 1999.

[113] S.R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. VARIUS: A Model of Process Variation and Resulting Timing Errors for Microarchitects. In *IEEE Transactions on Semiconductor Manufacturing*, Feb. 2008, 21(1):3-13.

[114] J. Sartori and R. Kumar, Distributed Peak Power Management for Many-core Architectures, In *Design, Automation, and Test in Europe (DATE)*, April 2009.

[115] R. Sasanka, S.V. Adve, Y. Chen, and E. Debes. The Energy Efficiency of CMP vs. SMT for Multimedia Workloads. In the *18th International Conference on Supercomputing (ICS),* June 2004, pp. 196-206.

[116] E. Schuchman and T.N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In the *32nd International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 160-171.

[117] Semiconductor Industry Association. International Technology Roadmap for Semiconductors – 2003 Edition. In http://www.itrs.net/links/2003itrs/home2003.htm.

[118] J.S. Seng, D.M. Tullsen, and G.Z.N. Cai. Power-Sensitive Multithreaded Architecture. In the *International Conference on Computer Design (ICCD)*, Sept. 2000, pp. 199-206.

[119] J. Sharkey, A. Buyuktosunoglu, and P. Bose. Evaluating Design Tradeoffs in On-Chip Power Management for CMPs. In the *International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2007, pp. 44-49.

[120] D. Shelepov, J.C.S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems, In the *ACM SIGOPS Operating Systems Review*, April 2009, pp. 66-75.

[121] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In the *30th International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 336-347.

[122] P. Shivakumar, S.W. Keckler, CR. Moore, and D. Burger. Exploiting Microarchitectural Redundancy for Defect Tolerance. In the *International Conference on Computer Design (ICCD)*, Oct. 2003, pp. 481-488.

[123] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In the *XII International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006, pp. 73-82.

[124] K. Skadron. Hybrid Architectural Dynamic Thermal Management. In *Design, Automation and Test in Europe (DATE)*, Feb. 2004, pp. 10-15.

[125] K. Skadron. Personal Communication, June 2006.

[126] K. Skadron, M.R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. In the *30th International Symposium on Computer Architecture (ISCA)*, June. 2003, pp. 2-13.

[127] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In the *31st International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 276-287.

[128] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In the *34th International Conference on Dependable Systems and Networks (DSN)*, June 2004, pp. 177-186.

[129] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In the *32nd International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 520-531.

[130] K. Stavrou and P. Trancoso. Thermal-Aware Scheduling for Future Chip Multiprocessors. In the *EURASIP Journal on Embedded Systems*, 2007.

[131] D. Sylvester, D. Blaauw, and E. Karl. ElastIC: An Adaptive Self-Healing Architecture for Unpredictable Silicon. In *IEEE Design and Test of Computers*, Nov.-Dec. 2006, pp. 484-490.

[132] D. Tarjan, S. Thoziyoor, and N.P. Jouppi. CACTI 4.0. *HP Laboratories Palo Alto Technical Report HPL-2006-86*, 2006.

[133] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Mitigating Parameter Variation with Dynamic Fine-Grain Body Biasing. In the *40th International Symposium on Microarchitecture (MICRO)*, Dec. 2007, pp. 27-42.

[134] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In the *35th International Symposium on Computer Architecture (ISCA)*, June 2008, pp. 363-374

[135] A. Tiwari, S.R. Sarangi, and J. Torrellas. ReCycle: Pipeline Adaptation to Tolerate Process Variation. In the *34th International Symposium on Computer Architecture (ISCA)*, June 2007, pp. 323-334.

[136] D. Tullsen, S. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In the *23rd International Symposium on Computer Architecture (ISCA)*, May 1996, pp. 91-202.

[137] Y. Wang, K. Ma, and X. Wang, Temperature-Constrained Power Control for Chip Multiprocessors with Online Model Estimation, In *36th International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 314-324.

[138] J.A. Winter and D.H. Albonesi. Addressing Thermal Non-Uniformity in SMT Workloads. In *ACM Transactions on Architecture and Code Optimization (TACO)*, May 2008, 5(1): 4:1-4:28.

[139] J.A. Winter and D.H. Albonesi. Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures. In the *38th International Conference on Dependable Systems and Networks*, June 2008, pp. 42-51.

[140] J.A. Winter and D.H. Albonesi. The Scalability of Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures. In the *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA)*, June 2008, pp. 55-62.

[141] M. Yilmaz, S. Ozev, and D.J. Sorin. Low-Cost Run-Time Diagnosis of Hard Delay Faults in the Functional Units of a Microprocessor. In the *25th International Conference on Computer Design (ICCD)*, May 2007, pp. 317-324.

[142] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. *The University of Virginia, Department of Computer Science, Technical Report CS-2003-05*, March 2003.

[143] V. Zyuban and P. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. In *IEEE Transactions on Computers*, March 2001, 50(3):268-285.