The Pennsylvania State University
The Graduate School

DYNAMIC RESOURCE MANAGEMENT FOR
ENERGY-EFFICIENCY AND QUALITY-OF-SERVICE IN CHIP
MULTIPROCESSORS

A Dissertation in
Computer Science and Engineering
by
Yang Ding

© 2010 Yang Ding

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2010
The dissertation of Yang Ding was reviewed and approved* by the following:

Mahmut Kandemir  
Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Padma Raghavan  
Professor of Computer Science and Engineering

Mary Jane Irwin  
Professor of Computer Science and Engineering

Qian Wang  
Associate Professor of Mechanical Engineering

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

Chip multiprocessors (CMPs) are becoming increasingly popular as performance improvements brought by increasing clock frequency alone are approaching their limits. Other factors, such as ease of verification and validation of individual cores (as compared to complex uncore architectures) and the ability to exploit both thread level (coarse grain) and instruction level (fine grain) parallelism, also boost trends towards chip multiprocessing.

While CMPs have already made their way into the commercial market, software support for CMPs is still in its infancy, and is expected to be the main roadblock to the effective use of CMPs. As the number of processor cores is expected to keep increasing, how to fully utilize the abundant computing resources on CMPs becomes a critical issue. Two possible ways of exploiting CMPs include a single application scenario and a multiple application scenario. In the first case, the entire CMP is dedicated to a single application at a time. This option can be effectively exploited by applications that are getting increasingly complex and data intensive (particularly large codes from scientific computing, database and embedded image/video processing domains), unless the number of cores on the chip is increased beyond a certain count. In the multi-application scenario, which is also called multi-tasking, multiple (independent) applications are executed on the CMP at the same time. This is expected to be an important alternative at least in the short term especially for applications that have limited parallelism.

Parallelization has been studied for many years since the invention of the first parallel machine. However, there are still a lot of open questions to be solved such as compiler-based automatic parallelization. The multi-core era brings new challenges into the research scope of parallel architecture and applications. For example, resources on a single chip such as caches and inter-connections are now shared by multiple processing elements, which is not the case for traditional parallel or distributed systems.
Motivated by these observations, this dissertation work focuses on how to adapt application executions dynamically to improve the energy-efficiency and quality-of-service by utilizing the application level characteristics in the resource management. There are two major reasons to study the dynamic adaptations of applications. First, an application can have different characteristics and computing requirements during different phases of computations. Second, future computer architecture will have parameter variations and heterogeneity due to process variations or heterogeneous system design. My work investigates dynamic application adaptations, the partition of processors cores among multiple applications, and different thread scheduling schemes for both threads of the same application and threads across concurrently-running independent applications.

Performance has been the most important metric for computing in the past. This has been changed recently as optimization metrics other than performance are becoming increasingly important. These metrics include availability and energy efficiency among others. In many execution scenarios where CMPs are involved, satisfying multiple metrics (e.g., achieving high availability and low energy consumption) can be critical. In the first half of this dissertation, several approaches are studied to adapt the application executions at runtime in order to improve the energy efficiency. Different metrics are used with an emphasis on the tradeoff between performance and energy consumptions. Helper threads are proposed to collect characteristics of the application execution threads and to determine the ideal resource allocation schemes.

Managing quality of service (QoS) and providing service differentiation have drawn a lot of research interest recently, especially as CMPs become prevalent and virtualization environments are widely deployed. The lack of quality assurance on CMPs has become a major concern because concurrently-executing applications or even threads for the same application can compete arbitrarily on shared resources such as cores, shared caches, and off-chip memory bandwidth. Various hardware resource partitioning/reservation schemes have been discussed in the literature. In the second half of this dissertation, a software approach is investigated for QoS by dynamic tuning the time-slices of multi-application workloads in time-sharing operating systems. In order to satisfy the QoS requirements specified by users, a formal feedback control framework is built to dynamically tune the process nice values and associated CPU time-slices for simultaneously running applications. Experimental results show that the proposed framework can successfully track the quality of service targets and provide service differentiation.
# Table of Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## Chapter 1
The Multicore Era and New Challenges in Resource Management 1
1.1 Dynamic Changes in Application Behavior and Hardware Resources 3
1.2 Resource Management for Energy Efficiency 4
1.3 Resource Management for Quality of Services 5

## Chapter 2
Adapting Application Execution in CMPs Using Helper Threads 7
2.1 CMP Architecture and Target Scenario 9
2.2 Helper Thread Based Approach to Core Availability Problem 14
2.3 Experimental Setup 17
2.3.1 Applications 17
2.3.2 Simulation Setup 19
2.4 Experimental Results 22
2.4.1 Results without re-threading 22
2.4.2 Results with re-threading 24
2.4.3 Breaking Down the EDP Reductions 27
2.4.4 Results with DVFS 29
2.4.5 Sensitivity Analysis 29
2.4.6 Limited Number of Configurations 31
# 2.4.7 More General Variations ........................................ 32
# 2.4.8 Discussion on Overheads and Limitations ....................... 34

2.5 Related Work ...................................................... 35
2.6 Conclusions ....................................................... 38

## Chapter 3
Dynamic Core Partitioning for Energy Efficiency 39

3.1 Target Architecture, Applications,
Optimization Metric, and Execution Steps ......................... 41
3.2 Empirical Motivation ............................................. 43
3.3 Our Dynamic Core Partitioning Scheme .......................... 45
3.4 Experimental Setup and Results .................................. 51
3.4.1 Alternative Scheduling ........................................ 54
3.4.2 Higher Number of Cores ....................................... 56
3.4.3 Idle Core Repartitioning ....................................... 57
3.4.4 Different Executing Patterns .................................. 58
3.4.5 More on Executing Steps ...................................... 58
3.4.6 Time Sharing vs Space Sharing ................................. 59
3.5 Related Work ...................................................... 60
3.6 Conclusions ....................................................... 61

## Chapter 4
QoS Aware Dynamic Time-Slice Tuning 63

4.1 Background and Motivation ....................................... 65
4.2 Dynamic Time-slice Tuning Controller ............................ 68
4.2.1 Fraction Controller ........................................... 68
4.2.2 Nice Controller ................................................ 70
4.3 Experimental Setup and Evaluation ................................ 74
4.3.1 Tracking the QoS Targets ...................................... 77
4.3.2 Selection of QoS Targets ....................................... 78
4.3.3 Extending to Multicore Architectures ......................... 79
4.4 Related Work ...................................................... 81
4.5 Conclusions ....................................................... 82

## Chapter 5
Future Work and Conclusions 83

5.1 Future Work ...................................................... 83
5.1.1 Heterogeneous Systems ....................................... 83
5.1.2 Beyond Single Level Resource Management ................... 85
5.1.3 Other Objectives and Metrics .................................. 86
5.2 Summary of Contributions ......................... 87

Bibliography ........................................... 88
## List of Figures

2.1 CMP architecture considered in our work. L1 instruction and data caches are private, whereas the unified L2 cache is shared across all cores. ................................................................. 9

2.2 Illustration of reduced core availability. (a) When the two (rightmost) core become unavailable, their threads are migrated (re-mapped) to available cores. (b) When using fewer cores, unused ones can be turned off, potentially leading to a lower EDP value. . . 10

2.3 Example delay and EDP curves. .................................................. 11

2.4 Illustration of how we change the number of cores and threads to cope with reduced core availability. The notation of \((a, b)\) represents execution with \(a\) threads running on \(b\) cores. ........................................ 13

2.5 One-dimensional (curve) fitting. Both curves capture the predicted EDP values. The curve fitting model is updated as more data points are collected at runtime. .................................................. 16

2.6 Two-dimensional (surface) fitting. Data points in the lower triangle can be used to predict the next configuration to minimize the EDP. 16

2.7 EDP values for different steps in the FT, MG, LU, and CG benchmarks. Each curve captures the normalized EDP values for one application step when executed with 16 threads but different number of cores. Steps in FT, LU, and CG exhibit similar behavior; whereas the steps in MG fall into several groups. .................. 21

2.8 One-dimensional fitting results for FT, MG, LU, and CG. A bar corresponding to a value of \(p\) on the x-axis, gives the normalized EDP value when the number of cores drops from 16 to \(p\). For our curve fitting approach (denoted 1-D fitting), the value within the parentheses indicate the number of initial data points. .......... 23

2.9 One-dimensional fitting snapshot. .............................................. 25

2.10 Two-dimensional fitting snapshot. ............................................. 25
2.11 Two-dimensional fitting results for FT, MG, LU, and CG with 20 initial data points. All bars are normalized to the first bar in each group of bars. The initial data points are randomly selected and results shown here are based on 1,000 experiments. 

2.12 Energy and delay breakdown in one-dimensional fitting for FT. All values are normalized with respect to those when no configuration changes are made.

2.13 Energy and delay breakdown in two-dimensional fitting for FT. All values are normalized with respect to those when no configuration changes are made.

2.14 Summary of the energy and delay breakdown in one-dimensional fitting with 4 initial data points for MG, LU, and CG. All values are normalized with respect to those when no configuration changes are made.

2.15 Summary of the energy and delay breakdown in two-dimensional fitting with 20 randomly selected initial data points for MG, LU, and CG. All values are normalized with respect to those when no configuration changes are made.

2.16 One-dimensional fitting with the different number of initial data points. We assume that the number of available cores drops from 16 to 14 and the thread count remains at 16.

2.17 Two-dimensional fitting with the different number of initial data points. We assume that the number of available cores drops from 16 to 14.

2.18 One-dimensional fitting for a general core availability variation pattern.

2.19 Two-dimensional fitting results for a general core availability variation pattern.

2.20 Difference in the configurations chosen by our two-dimensional fitting scheme and the optimal scheme. Each group of bars indicate the difference in thread count, core count, and EDP value achieved. The EDP value differences are normalized to the optimal values, whereas the differences in thread/core count are in absolute terms.

3.1 (a) The number of cores allocated to FT and MG using a dynamic allocation scheme that employs exhaustive search. (b) The resulting EDP values over execution steps normalized to values when using the equal partitioning scheme.

3.2 The main components of our proposed dynamic core partitioning approach.
3.3 An example core partitioning using our scheme when four applications (A, B, C, and D) are executing. Each box represents an execution step. Allocated core counts are specified inside the boxes.

3.4 Pseudo code for the runtime model and the OS partitioner in our dynamic core partitioning scheme.

3.5 Core partitioning for LU and MG with our dynamic partitioning scheme on a 16-core CMP.

3.6 W-EDPG for two-application workloads with our dynamic partitioning scheme on a 16-core CMP.

3.7 W-EDPG for four-application workloads with our dynamic partitioning scheme on a 16-core CMP.

3.8 Core partitioning for FT, MG, CG and LU with our dynamic partitioning scheme on a 16-core CMP.

3.9 Core partitioning for FT, MG, CG and LU with core reservation enabled in our dynamic partitioning scheme on a 16-core CMP.

3.10 W-EDPG for four-application workloads with our dynamic partitioning scheme on a 32-core CMP.

3.11 W-EDPG for different workloads with our dynamic partitioning scheme and modified equal partitioning scheme on a 16-core CMP.

3.12 Core partitioning of a dynamic start/exit workload using our partitioning scheme with core reservation on a 16-core CMP.

4.1 Illustration of IPC variations and actual CPU fractions when using different nice values: (a)&(d) default nice values; (b)&(e) bzip2 with nice value -5; (c)&(f) bzip2 with nice value 5.

4.2 Feedback control loop framework in controlling IPC.

4.3 Illustration of the correlation coefficient between IPC values and CPU fractions for four applications when executing them together.

4.4 CPU fractions with different base time-slices for bzip2 and milc when executing them concurrently.

4.5 CPU fractions when executing a workload of two I/O-intensive applications.

4.6 (a) IPC variations, (b) actual CPU fractions, and (c) nice values assigned when targeting IPC values of bzip2 and gcc at 5% and 10%.

4.7 (a) IPC variations, (b) actual CPU fractions, and (c) nice values assigned when targeting IPC values of both bzip2 and gcc at 8%.

4.8 IPC values for bzip2 and milc in the default scenario.

4.9 IPC values after applying the controller for bzip2 and milc.

4.10 IPC values for bzip2, gcc and libq in the default scenario.

4.11 IPC values for bzip2, gcc and libq after applying our controllers.
4.12 IPC values for a four-application workload in the default scenario. . 76
4.13 IPC values for a four-application workload after applying our controllers. .......................................................... 76
4.14 Feedback control framework for multiple multi-threaded applications on multicore. ........................................ 80
4.15 IPC values of the bodytrack thread on core 0 with IPC target at 0.1. 81

5.1 An example of CMP architecture with heterogeneity. Each row corresponds to a bin, with cores having the same power/performance characteristics. Shadings correspond to the core frequencies of the bins. .......................................................... 84
5.2 Normalized EDP values when using different mapping methods. Each group of bars is for a set of 4 applications running simultaneously on the same CMP. .................................................. 84
List of Tables

2.1 Our major simulation parameters and their default values. . . . . . 20
3.1 Major simulation parameters. . . . . . . . . . . . . . . . . . . . . . 51
4.1 Major simulation parameters. . . . . . . . . . . . . . . . . . . . . . 75
4.2 Selected SPEC2006 benchmarks. . . . . . . . . . . . . . . . . . . . 76
Acknowledgments

I want to first thank my thesis advisor, Professor Mahmut Kandemir, for guiding and supporting me throughout the past years. He helped me build a solid knowledge foundation and develop the skill sets for research. I will not forget all the hand-written marks he made on my paper drafts, not mention the moments when he sit besides me and explained to me sentence by sentence on how to write research papers. Professor Kandemir sets a life-time model for me to dream big, to work hard, to think smart, and to keep challenging oneself.

I was very fortunate to participate the research work in PxP (Performance x Power) group, which was led by Professor Mary Jane Irwin and Professor Padma Raghavan. Professor Irwin has kept me updated with the cutting-edge research activities in both academia and industry and given me opportunities to meet and learn from world-famous researchers. Professor Raghavan taught me many things from how to give presentations to various scientific mathematical algorithms. I want to thank them for everything they have done for me. My sincere gratitude goes to Professor Qian Wang for serving in my thesis committee. Professor Wang is always patient and willing to help. She also identified several important mistakes I overlooked and directed me to improve my work.

As a proud member of Microsystems Design Lab (MDL), I want to thank all the former and current MDLers who are unselfish to share ideas and supportive to each other. Although I do not have space to list everyone that I am grateful to, my special thanks go to Guangyu Chen, Guilin Chen, Feihui Li, and Xiaoxia Wu.

Last but not least, I want to thank my family and friends for their endless love, enormous sacrifice, and continuous encouragement over the years. My wife, Yao Wang, has helped me a lot in almost everything, from making everyday meals to creating complicated Excel figures. I cannot imagine life without her. I am indebted to my parents who have shown great patience and confidence for my study. They have invested as much as they could on me and never asked me for anything other than doing my best in whatever I choose to do. My life would not be as colorful and enjoyable without my dear friends throughout the world.

xiii
Chapter 1

The Multicore Era and New Challenges in Resource Management

Computer industry has been developing at an unprecedentedly fast speed over the past decades by following the Moore’s law and enjoying all the benefits coming with it. Moore’s law says that the number of transistors that can be integrated on a die doubles every 18 months. With the ever-increasing transistor budget, hardware designers have more resources to utilize and more hardware modules can be implemented on microprocessors. Therefore, we have seen more and more complex processors over the years with deeper pipelines, wider issue logic and execution units, larger on-chip caches, more precise branch predictors and other advanced hardware techniques. As a result, the progress in process technology has been able to make the maximum clock frequency double approximately every two years until recently.

With all the innovations and improvements in design and manufacturing, people have got used to having a new processor which is twice faster every one and a half years to two years. Such growth had remained from Intel 8086 to 80486, from Pentium to Pentium 4 until Intel canceled its plan for the highest frequency processor in 2004. Soon after that, the industry made a new move from increasing the speed of single microprocessor to integrate multiple processor cores on one chip. There are already dual/quad/octo cores on the general microprocessor market (e.g., Intel’s dual-core Montecito [1] and quad-core i7[2], AMD’s quad-core Barcelona [3], Sun’s eight-core Niagara [4]), with more aggressive configurations
being shipped (e.g., Tilera’s 64-core TILE64 [5]) or being prototyped (e.g., Intel’s 80-core TeraFlop [6]).

There are many reasons why chip multiprocessors (CMPs) are becoming increasingly popular and has brought us into the multicore era. Firstly, the performance improvements resulting from increasing clock frequency alone are approaching their limits. Secondly, the power consumption keeps growing with the complexity of processor which not only consumes more power/energy consumption but also makes thermal hot spots to appear more frequently. Thirdly, as compared to complex single-core architectures, multicore brings the ease of verification/validation of individual cores. The design and manufacturing of multiple simple cores are more cost-effective than those for a complex microprocessor. Last but not least, the ability to exploit both thread level (coarse grain) and instruction level (fine grain) parallelism, also boost trends towards chip multiprocessing.

While CMPs have already made their way into the commercial market, software support for CMPs is still in its infancy, and is expected to be the main roadblock to the effective use of CMPs [7]. As the number of processor cores is expected to keep increasing, how to fully utilize the abundant computing resources on CMPs becomes a critical issue. Previously, with the doubling of processor frequency, software can obtain speedup without any modifications. If the application is computation bounded, the performance could get doubled with twice higher clock frequency. However, no such “free lunch” exist any more in the multicore era. Software must be written to take advantage of multiple threads/cores in order to scale the performance in a linear fashion.

Most programs implemented in the past are either sequential or explicitly threaded. A serial program may not run faster on multicore than a single processor, which is why customers may not see any improvement in execution their applications after installing a dual-core processor. Explicitly threaded applications are usually implemented assuming a specific architecture such as certain number of threads/cores and may not perform well on a different type of architecture. The future of performance scaling relies on scalable concurrent applications which can obtain significant benefits from most if not all the architectures that the software may be run on. Unfortunately, programming in this category is difficult and there are only a few applications with abundant parallelism in embedded system and
high performance computing domains. New programming models have been proposed recently, such as CUDA [8] and OpenCL [9], which help developers to create new applications or modify existing ones to take advantage of multicore architectures especially GPU accelerators. However, these new languages have not been mastered by most programmers and there are many legacy codes to execute.

These observations lead to two possible ways of exploiting CMPs, including a single application scenario and a multiple application scenario. In the first case, the entire CMP is dedicated to a single application at a time. These applications are getting increasingly complex and data intensive (particularly large codes from scientific computing, database and embedded image/video processing domains). They can effectively exploit this option, unless the number of cores on the chip is increased beyond a certain count. In the multi-application scenario, which is also called multi-tasking, multiple (independent) applications are executed on the CMP at the same time. This is expected to be an important alternative at least in the short term especially for applications that are lack of parallelism.

Besides the software challenge, the multicore architectures bring new challenges into the resource management of parallel architectures and applications. For example, resources on a single chip such as caches and inter-connections are now shared by multiple processing elements, which is not the case for traditional parallel or distributed systems. In order to manage the resources effectively, we need to consider both the application level information and the underlying hardware level information.

1.1 Dynamic Changes in Application Behavior and Hardware Resources

Applications have various computing requirements for their executions. For instance, the release of computer games usually comes with a list of basic hardware requirements and suggested hardware configurations. Consequently, different applications often behave differently on the same architecture. For example, one application may have a lot of L2 cache misses while the other can fit all the data in L1 caches. In fact, even within one application, the execution characteristics
may change as the execution continues. There could be program phases that are memory-bounded (i.e., majority of the execution time is for memory accesses) and other phases that are CPU-bounded (i.e., execution time is dominated by the calculations in the CPU). Therefore, an application can be optimized by dynamically changing the resource allocated to it.

On the other hand, the hardware resources may also change dynamically. For example, the number of available cores may change at runtime due to several reasons such as a pending thermal emergency, increase/decrease in transient errors, or as a result of an operating system (OS) decision. In such situations, if we expose the hardware changes to software level, applications can adapt to those changes by using techniques such as rethreading or thread migration.

Another new question is brought by process variations or heterogeneous system designs. As process technology moves into the deep sub-micron regime, it is becoming increasingly difficult to control critical transistor parameters such as gate-oxide thickness, channel length, and dopant concentration. As a result, these parameters may have different values than nominal, which may, in turn, lead to both power and timing variations across identically-designed components. In the context of CMPs, such variations may mean that different cores can have different power and performance characteristics (also called core-to-core variations). Similar situations exist in heterogeneous systems such as IBM Cell [10], where different types of processor cores exist. When applications are mapped to systems with process variations or heterogeneity, it would be ideal to let the applications choose which kind of cores to use based on their requirements.

### 1.2 Resource Management for Energy Efficiency

Performance has been the most important metric for computing in the past. This has been changed recently as optimization metrics other than performance are becoming increasingly important. These metrics include power consumption, energy consumption, availability and reliability among others. In many execution scenarios where CMPs are involved, satisfying multiple metrics (e.g., achieving high availability and low energy consumption) can be critical. In the dissertation work, I will use and propose different metrics with an emphasis on the tradeoff
between performance and energy consumptions. The first half of my dissertation aims to manage the resources efficiently both in performance and power/energy consumption.

My work focuses on how to adapt applications dynamically to the underlying CMP architectures and how to utilize the application characteristics to improve the resource management on CMPs. I also investigate the partition of processors cores among multiple applications, and different thread scheduling schemes for both threads of the same application and threads across concurrently-running independent applications. As future computer architecture will have parameter variations and heterogeneity due to process variations or heterogeneous system design, I will discuss potential future work in these areas with a brief example of study on thread allocations and thread-to-core bindings.

1.3 Resource Management for Quality of Services

Managing quality of service (QoS) and providing service differentiation have drawn a lot of research interest recently, especially as CMPs become prevalent and virtualization environments are widely deployed. The lack of quality assurance on CMPs has become a major concern because concurrently-executing applications or even threads for the same application can compete arbitrarily on shared resources such as cores, shared caches, and off-chip memory bandwidth. Various hardware resource partitioning/reservation schemes have been discussed in the literature. In the second half of my dissertation, I investigate a software approach for QoS by dynamic tuning the time-slices of multi-application workloads in time-sharing operating systems.

In most operating systems, the actual CPU time-slice for a process is based on a base time-slice and some dynamic adjustments. The base time-slice is the same for processes within the same priority level. The process priority and associated time-slice assigned by the operating system (OS) scheduler can usually affect its performance. When multiple applications compete for the CPU cycles in a round-robin manner, longer CPU time-slice often results in larger fraction of
CPU execution and leads to performance speedup for an individual application. In order to satisfy the QoS requirements specified by users, a formal feedback control framework is built to dynamically tune the process nice values and associated CPU time-slices for simultaneously running applications. Experimental results show that the proposed framework can successfully track the quality of service targets and provide service differentiation.
Adapting Application Execution in CMPs Using Helper Threads

In addition to the move into multicore era in architecture domain, applications are getting increasingly complex and data intensive (particularly large codes from scientific computing, database and embedded image/video processing domains). One of the interesting problems in this context is to adapt application execution to varying hardware resources in a performance and energy efficient manner, that is, to use the available resources carefully to achieve both good performance and low energy consumption.

In this chapter, we study the problem of how application executions can adapt to processor core availability change. Our goal is to decide – at runtime – the best strategy to employ when the number of cores available to an application is changed, considering the energy-delay product (EDP). Energy-delay product (EDP) is an important metric as it captures our desire of both achieving high performance and reducing energy consumption, both of which are critical in data intensive computing. In other words, we want to adapt the execution to processor core availability change with the goal of minimizing the EDP.

The approach proposed in this chapter employs a helper thread for this purpose. More specifically, we use a helper thread whose primary job is to collect – using performance counter information (or data) provided by the CMP architecture and a power model – energy-delay product statistics during the course of execution and decide the most appropriate number of cores, number of threads, and volt-
age/frequency levels to use when a variation on resource availability occurs. In making these decisions, the helper thread uses curve fitting and data interpolation methods [11]. Assuming long application execution time, using such a helper thread, our approach can collect a sufficient number of data points (core count, thread count, voltage/frequency level, and EDP value) at runtime. Based on these data points, it can accurately predict the behavior of the application and can then select the core count, thread count and voltage level to use when the number of available cores dynamically changes at runtime.

We implemented our approach using a full system simulator [12] and conducted experiments using four data-intensive applications: Fourier Transform (FT), Multi-Grid (MG), LU decomposition (LU), and Conjugate Gradient (CG). In our experimental evaluation of the proposed approach, we tested its three variants: (1) using a fixed number of threads and a fixed voltage/frequency level (i.e., adapting only the number of cores); (2) adapting both core count and thread count (under a fixed voltage/frequency level); and (3) adapting the core count, thread count and voltage level together in a coordinated fashion. The experimental results we collected indicate that our proposed helper thread based dynamic adaptation scheme is very successful in practice even if all the overheads brought by the helper thread are accounted for. Specifically, we are able to reduce the EDP value in scenarios (1), (2), and (3) mentioned above by as much as 21.7%, 54.6%, and 66.3% in FT, 35.5%, 77.2%, and 83.3% in MG, 44.2%, 88.2%, and 93.5% in LU, and 35.4%, 92.1%, and 94.2% in CG. The corresponding average EDP improvements under scenarios (1), (2), and (3) are 7.4%, 26.3%, and 46.1% in FT, 15.3%, 59.7%, and 70.5% in MG, 6.3%, 77.2%, and 83.4% in LU, and 14.2%, 91.0%, and 91.2% in CG. In addition, we also compared our approach to an optimal – but unimplementable – adaptation strategy. The collected statistics with our helper thread based approach and the optimal adaptation scheme reveal that, on average, we are within 5.9% of the optimal. Overall, our results indicate that, in order to minimize the EDP in a CMP based architecture, we need to select the number of cores, number of threads, and voltage/frequency levels very carefully, and a helper thread can be very useful for this purpose. Note that, while we present our analysis using four applications, our approach is quite general and can be used for other application domains as well.
2.1 CMP Architecture and Target Scenario

The CMP architecture considered in this work is of the type shown in Figure 2.1. There are \( n \) processor cores in total. Each core has its own private instruction and data L1 caches, and all cores share a unified on-chip L2 cache. Note that several commercial and academic CMP architectures discussed in literature fit into this template [13, 14, 15], although future CMPs are likely to have a distributed and banked L2 cache accessed through an inter-connection network. We assume that, if the application is not using a core, that core and its associated L1 caches can be turned off to save power. As leakage power is becoming an increasingly important component of the overall power budget [16], turning off a core and its L1 caches can reduce the chip-wide power consumption dramatically. Several architectural techniques [17, 18] can be used for turning off L1 caches. We further assume that the cores in our CMP can be voltage/frequency scaled. Scaling down the frequency of a core increases the length of its clock period (which in turn increases application execution latency), whereas scaling down its voltage reduces power consumption. Therefore, interesting power-performance tradeoffs can be studied by playing with the voltages/frequencies of cores in a CMP environment.

![Figure 2.1. CMP architecture considered in our work. L1 instruction and data caches are private, whereas the unified L2 cache is shared across all cores.](image)

The major metric we use in this work is the energy-delay product (EDP) [19], which captures the desire of both achieving high performance and reducing energy consumption. EDP can be calculated based on the following equations:

\[
\begin{align*}
EDP &= \text{Energy} \times \text{Delay}, \\
\text{Energy} &= \text{Dynamic Energy} + \text{Leakage Energy} \\
&= \text{Dynamic Power} \times \text{Delay} + \text{Leakage Power} \times \text{Delay}.
\end{align*}
\]
Both energy consumption and the delay of program execution are lower-the-better metrics. The total energy consumption consists of dynamic energy and leakage energy. Dynamic energy is proportional to $CV_{dd}^2$, where $C$ is the total capacitance and $V_{dd}$ is the supply voltage. Leakage power is proportional to $I_{leak}V_{dd}$. Here, $I_{leak}$ is the leakage current, which mainly consists of subthreshold and gate leakage.

Figure 2.2. Illustration of reduced core availability. (a) When the two (rightmost) core become unavailable, their threads are migrated (re-mapped) to available cores. (b) When using fewer cores, unused ones can be turned off, potentially leading to a lower EDP value.

Figure 2.2(a) illustrates the execution scenario we focus in this chapter. In this scenario, an application executing on a CMP (say, for simplicity, $n$ threads running on $n$ cores) is informed during its execution that one or more of the cores it currently uses are about to be taken away from it. This can be due to several reasons such as a pending thermal emergency, increase in transient errors, or as a result of an operating system (OS) decision. The approach described in this chapter is applicable to all these scenarios; in fact, the actual reason behind such availability change is orthogonal to the main focus here. Note that the core availability can change in the other direction as well (i.e., the number of available cores to an application can increase at runtime). Our approach handles the increased or reduced core availability with a general scheme. Therefore, without loss of generality, we assume that the OS decides (based on its global resource management policies) to take away some of the cores on which the application is running. In this case the execution should somehow adapt to this new condition. As our focus is not on techniques for failure recovery, issues of thread imaging and restart after core failure is beyond the scope of this chapter. In the rest of this section, we discuss the different adaptation options that can be considered:
• The first, and the most intuitive option, is to let the OS re-map (migrate) the threads that were originally running on the cores to be taken away to available cores. For example, if \( m \) of the original \( n \) cores become unavailable, the threads on these \( m \) cores can be migrated to the remaining \( n - m \) available cores. While many types of migration (re-mapping) schemes can be employed for this purpose, the different re-mappings can lead to dramatically different results. Figure 2.2(a) depicts a specific migration scheme with \( n = 16 \) and \( m = 2 \). In this migration scenario, \( m \) of the \( n - m \) available cores take the \( m \) orphan threads, and this leads to an increase in their workload.

![Figure 2.3. Example delay and EDP curves.](image)

• While trying to use all available \((n - m)\) cores may be reasonable from the performance perspective alone, it may \textit{not} be the best option when we consider the EDP. More specifically, \textit{is it possible to use fewer cores than \( n - m \) and achieve a lower energy-delay product?} This may be possible as the unused cores (and their L1 caches) can be turned off, as explained earlier, to save leakage energy. Meanwhile, such a reduction in the number of cores and L1 caches used does not always significantly affect the performance. As a potential scenario, based on the thread migration scheme given in Figure 2.2(a) and omitting the potential changes in cache behavior due to core/L1 turnoff, one can expect the execution delay and EDP curves to be as shown in Figure 2.3 (assuming \( n \) is 16). The main observation from this plot is that the large drops in latency (due to parallelization) occur only in certain core counts. In fact, the execution delay curve drawn in Figure 2.3 follows \( \left\lceil \frac{16}{\text{number of cores}} \right\rceil \). Therefore, as an example, if \( n = 16 \) and two cores become unavailable (i.e., \( m = 2 \)), we may work with \( q = 8 \) cores (instead of \( n - m = 14 \)).
cores) as the latencies with these two configurations are expected to be similar (see Figure 2.2(b)), save for the cache behavior. In addition, since 14 cores will have higher power consumption than 8 cores, EDP with 8 cores will be lower than that with 14 cores, as also illustrated in Figure 2.3 (in fact, as we increase the core count from 8 to 14, the EDP value continuously increases without considering cache impacts). Note however that 14 cores have more aggregate on-chip L1 cache capacity, and as a result, in reality using 14 cores may result in lower execution latency than using 8 cores. On the other hand, the performance overheads brought by operating system support such as synchronization can vary for systems with different number of cores. Therefore, in practice, the minimum EDP value could be caught with a core count of $q$, where $8 \leq q \leq 14$, as also depicted in Figure 2.3.

- So far, we implicitly assumed that, when $m$ cores become unavailable, we still try to continue execution with $n$ threads. Now, supposing that we are also able to obtain a version of the application that can use a different number of threads, we may want to use an $r$-thread version such that the total EDP value is further minimized. The important point here is the question of how such a version (re-threading) can be obtained. One option is dynamic code modification which can generate any desired version at runtime. However, this may be costly in some cases as compilation time required for re-threading contributes to the overall execution latency. Note that, for applications implemented using OpenMP [20], the cost of re-threading can be much less since the number of threads can be set at runtime using an environment variable [21] or by calling `omp_set_num_threads()` before each parallel region if the OpenMP runtime library supports. Another option would be to prepare, for each function of the application, a different version (that uses a different number of threads) beforehand at compile time and use (link) the appropriate multi-threaded version at runtime when the core unavailability takes place. This option, also called static versioning, is possible in general since we can expect the same set of functions to execute for many times in a large and long-running application. In practice, we may not need all possible versions of each function with all possible thread counts. Instead, we can do a reasonably good job with only a small set of versions (e.g., 2, 4, 8, and 16 thread versions) and switch to the appropriate version at runtime. Note that there are certain applications implemented with specialized threads where the number of threads
cannot be changed arbitrarily, though the applications we study do not belong to this category. An important issue at this point is the place in the code at which the new version will be invoked. Obviously, such an invocation cannot take place at any arbitrary point since this would require us to prepare versions considering all possible points where core unavailability can occur. Instead, one can adopt the following strategy: when the unavailability occurs, we select a suitable number of cores and continue executing the application without changing the number of threads until a function boundary is reached. When this boundary is reached, we change the number of threads as well. This approach is illustrated in Figure 2.4. In this example, if the core unavailability occurs in function $f_2$ at iteration $i$, (i.e., during the $i$th visit of the function), we can continue to use $n = 16$ threads on $q$ cores (where $q$ is the number of cores we select considering the EDP) until the return of $f_2$. When this happens, we start to use $r$ threads ($r \leq 16$) and execute them on $p$ cores (again $r$ and $p$ are selected based on the EDP). In this chapter, we use the notation of $(a, b)$ to represent an application execution using $a$ threads running on $b$ cores.

![Figure 2.4](image-url) Illustration of how we change the number of cores and threads to cope with reduced core availability. The notation of $(a, b)$ represents execution with $a$ threads running on $b$ cores.

- The results may be even better, from an EDP perspective, if dynamic voltage/frequency scaling is employed. Suppose that, based on the analysis outlined above, we decided to use $p$ cores and $r$ threads. It may be possible to reduce the EDP further by using $p'$ ($p' > p$) cores and $r'$ ($r' > r$) threads with the voltages and frequencies scaled down. In this work, we assume that all the cores use the same voltage/frequency level. That is, when voltage/frequency scaling is applied, their voltages/frequencies are scaled to the same value.

Clearly, one can expect the best (minimum) EDP value when the number of threads, number of cores and voltage/frequency level are selected carefully, considering the interactions among them. In the following, we explain and experimentally evaluate a helper thread based approach to this dynamic adaptation problem.
2.2 Helper Thread Based Approach to Core Availability Problem

In this section, we explain our helper thread based approach to the core availability problem. The motivation for helper threads comes from the observation that future CMPs will include a large number of cores and will execute applications that involve a large number of threads [22]. In such an execution environment, it is entirely feasible to allocate some of the cores and/or threads to various tasks that enable better application execution. For example, helper threads can be used to collect valuable statistics about application execution and execution environment, using performance counters and sensors. As another example, several recent efforts [23, 24] employed helper threads for prefetching data in cache memories. In a scenario with a large number of cores and threads, one can expect the additional costs brought by such helper threads to be more than compensated by the EDP benefits they bring.

In our context, we use a single helper thread which collects performance counter information and implements curve fitting to predict the ideal number of cores, threads, and voltage/frequency level to use when a variation in core availability occurs. We now explain the use of the helper thread under three different scenarios, each of them being more complex than the previous ones.

In the first scenario, we assume that the only control parameter is the number of cores. That is, in reacting to a variation in core availability at runtime, we can only change the number of cores; the number of threads and voltage/frequency levels are not changed (we use the current number of threads and the maximum voltage/frequency level). As explained earlier in Section 2.1, when $m$ of the original $n$ cores become unavailable to an application, it may not always be the best choice to use all $n - m$ available cores when considering the EDP as the primary metric of optimization. In this scenario, the functionality of the helper thread can be explained as follows. As the execution progresses, the helper thread collects statistics, with the help of performance counters and a power model, that keep track of the EDP values obtained under various number of cores experienced so far. When a variation on core availability occurs, the helper thread uses the data collected so far (which is essentially a set of [Number of cores, EDP Value]
pairs observed thus far in execution), and makes use of curve fitting to predict
the ideal number of cores to use to achieve the lowest EDP value. While our cur-
rent implementation uses a particular curve fitting scheme (piecewise cubic spline
interpolation [11]), the selection of the curve fitting scheme to employ is really or-
thogonal to the main focus of our approach. Note that, when an adaptation takes
place, the helper thread continues to record the EDP value observed and updates
its database to achieve better predictions in the future. It is important to point out
that some of the [Number of cores, EDP Value] pairs can also be obtained using
profiling or from prior executions of the same application. These pairs can then be
reused in the current execution for predictions. To summarize, our helper thread
implements two functionalities. First, it maintains the application’s EDP values
observed under different core counts. Second, it decides the next configuration to
use through curve fitting.

Figure 2.5 gives an example of the curve fitting in such scenario, assuming
two cores out of the original 16 cores become unavailable to our application at
some point during the course of execution. If we have initial points of EDP at
core counts of 2, 8 and 16, we can predict that the optimal point to operate is
with 14 cores as illustrated by the dotted curve in Figure 2.5. After the execution
of current function call, the energy-delay product can be calculated (using the
performance counters supported by the underlying CMP architecture) and used
as another data point (at core count of 14) for future curve fittings. This is
captured by the solid curve in Figure 2.5, and as a result, 9 cores are chosen to
continue the execution. That is, as more data points are collected, our curve fitting
model is updated. We explain the further details of this adaptation scheme in our
experimental evaluation.

In the second scenario, the control parameters we have are the number of cores
and the number of threads. That is, in reacting to a variation in core availability,
the helper thread selects a new core count and a new thread count. However,
as explained earlier in Section 2.1, thread count changes can only be performed
at function boundaries (our current implementation postpones the thread count
change to the next function boundary). As in the case of first scenario, the helper
thread makes its prediction using curve fitting. Clearly, the curve fitting in this case
is a two dimensional one (i.e., it is actually surface fitting) as it involves predicting
both thread and core counts, and our current implementation uses triangle-based linear interpolation for this purpose [11].

Figure 2.6 illustrates the two-dimensional space for the number of cores and the number of threads, assuming both of them are no more than 16. It is clear that using more cores than the number of threads can bring no additional performance gains but only wastes extra power. Therefore, the exploration space in Figure 2.6 is actually only the lower triangle portion. Ideally, we want to find the optimal point in the triangle (i.e., the best \((a, b)\) configuration), when re-threading is possible. Note that, if the number of threads is fixed as in the first scenario, the exploration translates into finding the optimal point along a certain vertical line in Figure 2.6. Therefore, the first scenario is just a special (and simpler) case of the second one.

In the third scenario, the control parameters include core count, thread count and voltage/frequency levels. That is, we also predict the new voltage/frequency level to use over the second scenario explained above. As stated earlier, in this study, all the cores use the same voltage/frequency level, but it is possible to extend our approach to cover cases where we have the flexibility of changing the voltage/frequency level of each core independent of the other cores. As in the pervious two scenarios, we use curve fitting (in this case it is three dimensional) to predict the best operating point. The particular curve fitting scheme used in this work is triangle-based linear interpolation.
It is important to note that the EDP values that are profiled beforehand or predicted at runtime in the curve fitting models are not for the complete execution of an application. Instead, they correspond to certain execution intervals (i.e., parts of the full application execution). In order to make the predictions accurate, the different execution intervals involved in a curve fitting model should behave similarly. Fortunately, we are able to extract and separate these execution intervals with some knowledge of the application, especially for most data-intensive applications that spend majority of their execution time in some loop iterations. For example, an execution interval can be a large function call or a program structure that is executed repeatedly. We discuss how to identify or define these intervals in details later in Section 2.3.1.

In all the three scenarios above, we use some profiled results as the initial data points in the exploration space. These results are profiled only when the execution intervals are executed for the first time. Therefore, the profiling cost is reasonably low. In our approach, the helper thread updates the prediction models at runtime using only what can be observed over the execution. We do not consider forcing the execution in order to explore more data points in the exploration space. Even though the runtime exploration may improve the predication accuracy, the system may be unnecessarily running with suboptimal EDP.

2.3 Experimental Setup

In this section, we briefly describe the benchmark applications used in this chapter and the simulator framework for our experiment study. Note that these applications will be considered in later chapters as well and the simulator framework with performance and power models is utilized throughout this dissertation.

2.3.1 Applications

Four applications are used in this chapter to evaluate our approach, including Fourier Transform (FT) and MultiGrid (MG), LU decomposition (LU), and Conjugate Gradient (CG). All of these programs are from the NAS Parallel Benchmark Suite [25]. We used their OpenMP implementations in version 3.2 with the class
W inputs [26]. These codes represent computations that are used in a large variety of modeling and simulation applications based on finite element, finite difference and spectral methods.

- Fourier Transforms are important in many domains such as digital signal processing and solving partial differential equations. Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. A Discrete Fourier Transform (DFT) of length $N$ can be represented as a sum of two DFTs of length $N/2$, and this process can be repeated recursively to obtain a divide and conquer algorithm. This partitioning significantly reduces the computational cost of the DFT, hence the term Fast Fourier Transform (FFT). The FT implementation we use has six major iterations, which altogether take 90% of the serial execution time. Each iteration performs the kernel of a three-dimensional FFT in function $fft$. This three-dimensional FFT kernel implements three one-dimensional FFT’s, corresponding to three function calls within each iteration ($cfft1$, $cfft2$ and $cfft3$). The behavior of these three functions are not the same but very similar.

- The MultiGrid (MG) method is typically used for solving elliptic partial differential equations on a discretized physical domain. The main idea behind MG is to represent the physical domain with a hierarchy of discretization from fine grain to coarse grain. The solution for the physical domain is obtained with extrapolation between coarser and finer grids. The MG implementation we use has four major iterations, which consume 70% of the total serial execution time. It computes the solution of the three-dimensional scalar Poisson Equation. Each loop iteration consists of the multigrid operation (function $mg3P$) and the residual calculation (function $resid$). The multigrid operation is performed with a sequence of different computations, implemented in functions such as $rprj3$, $psinv$, $interp$ and $resid$. Since there are a large number of function invocations within one iteration, we divide them into 7 function groups separately by each function call of $resid$. This partition is randomly chosen based on the source code structure. Other partitions are also possible. As a result, different function groups behave differently but the same function group has similar computations across iterations.

- LU decomposition is a matrix decomposition that translates a matrix into the product of lower and upper triangular matrices. Such decompositions are often
used in numerical analysis to solve systems of linear equations or calculate the determinant. The LU benchmark we use implements symmetric successive over-relaxation (SSOR) to factorize a set of equations into lower and upper triangular systems. Function \textit{ssor} consumes most execution time and its major loop has 300 iterations. Each iteration incurs many computations and performs both lower and upper triangular solutions. As its execution latency is much longer than the other chosen benchmarks, we focus on the first 15 iterations only.

- Conjugate Gradient is an iterative method which provides the numerical solution for systems of linear equations with symmetric and positive definite matrix. It can be applied to very large sparse systems which arise regularly when numerically solving partial differential equations. The CG benchmark implements a Conjugate Gradient method to approximate the smallest eigenvalue of a large, sparse, unstructured matrix. The main function has a major loop with 15 iterations which dominates the execution time. Each iteration calls the \textit{conj\_grad} function and implements the inverse power method.

We use the term of \textit{execution step} in the following discussion to indicate the granularity in which dynamic adaptations can be made. An execution step stands for an one-dimensional FFT function in FT, an function group in MG, or an major iteration in LU and CG. The source codes for the benchmarks are hand-instrumented so that the execution step boundaries can be identified at runtime.

### 2.3.2 Simulation Setup

We used the Simics 3.0 toolset [12] to perform our experiments. Simics is a multiprocessor simulator that can be used to perform full system simulation. The abstraction of the CMP architecture used in this study is given earlier in Figure 2.1. In our simulator, each core is based on SPARC architecture and runs unmodified Solaris 9 operating system with Sun Studio compilers and tools to support OpenMP [27]. Table 4.1 gives the major simulation parameters used in this study with their default values. In this work, (16, 16) corresponds to our default configuration before the core unavailability occurs (i.e., 16 threads running on 16 cores). If re-threading is not an option, after the unavailability, we use the configuration (16, x), where x < 16. If re-threading is possible, other configurations
are also considered. We assume that, in each of the four applications, the core unavailability takes place after the kernel computation starts (i.e., when entering the major loops). If $m$ cores become unavailable, configuration $(16, 16-m)$ is used as default to continue the execution. Our baseline (against which we compare our approach) is the case when the thread count and the core count do not change, and the remaining computations are executed using the default configuration of $(16, 16-m)$.

**Table 2.1.** Our major simulation parameters and their default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores ($n$)</td>
<td>16</td>
</tr>
<tr>
<td>Number of Threads</td>
<td>16</td>
</tr>
<tr>
<td>Highest Core Frequency</td>
<td>2GHz</td>
</tr>
<tr>
<td>Highest Voltage Level</td>
<td>1.1V</td>
</tr>
<tr>
<td>Number of Voltage/Frequency Levels</td>
<td>5</td>
</tr>
<tr>
<td>Core Issue Width</td>
<td>1</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64K, 2-way, 2 banks</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64K, 2-way, 2 banks</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>4MB, 16-way, 2 banks</td>
</tr>
<tr>
<td>Process Technology</td>
<td>70nm</td>
</tr>
<tr>
<td>L1 Access Latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L2 Access Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory Access Latency</td>
<td>260 cycles</td>
</tr>
</tbody>
</table>

Accurate estimation of the EDP values is critical in evaluating our work as we target at reducing the EDP. We rely on the various performance counters in Simics for this purpose, along with existing support from proper power models. To calculate the EDP value, we need to calculate the energy consumption and the application execution latency. The execution latency can be easily calculated using the number of cycles spent in execution (which can be obtained from performance counters) and the clock frequency used. The energy consumption we calculate include both leakage energy and dynamic energy components for cores, L1 caches, and shared L2 cache. We obtain the leakage power and dynamic power for the cache reads and writes from CACTI 5.0 tool [28] for both L1 and L2 caches. Together with the number of cache accesses (hits/misses) from performance counters in the full system simulator, we can estimate the energy consumption for caches. As for energy consumption for cores, we employ a power model similar to the one used in Wattch [29] and scale it appropriately to get dynamic and leakage power numbers separately.

Figure 2.7 presents the EDP values for the four benchmarks over different execution steps under the $(16, x)$ configurations. Each curve shows the normalized
Figure 2.7. EDP values for different steps in the FT, MG, LU, and CG benchmarks. Each curve captures the normalized EDP values for one application step when executed with 16 threads but different number of cores. Steps in FT, LU, and CG exhibit similar behavior; whereas the steps in MG fall into several groups.

EDP values for one execution step (a function, an iteration, or a function group) when the application runs with different number of cores. All the results are normalized with respect to the maximum EDP value observed. An important point to note from these results is that, the curves of EDP results over different steps are similar in FT, LU, and CG; but they differ more in the case of MG. This is because the functions in FT behave similarly (one-dimensional FFT) and computations in different loop iterations do not differ much in both LU and CG. In contrast, as mentioned earlier, we divide the functions in MG into different groups. As a result, the curves that belong to the steps in the same group are similar, but they exhibit a different trend from those falling into the different function groups.

This difference between MG and the other applications allows us to test the effectiveness of our helper thread based approach for two general types of applications. First, for applications that operate over similar tasks iteratively, a single model can be used to apply our curve fitting method and predict the next configuration to choose. Second, for applications with complex iterations (i.e., each
iteration calls different functions and executes for long periods of time), we can divide the complex iteration into several phases and build different models for different phases. Several studies [30, 31] can be used to identify application phases based on dynamic runtime characteristics. For simplicity, we partition a complex iteration in MG based on the static code structure in our study. Overall, the trends exhibited by the curves in Figures 2.7 indicate that curve fitting can be successful in predicting the next (most appropriate) configuration to employ.

2.4 Experimental Results

In our experiments, we start curve fitting with initial data points collected before the current execution. For example, in the FT application, we can measure the EDP of the first function call (for one-dimensional FFT) under different number of cores and different number of threads. Note that, we do not need a complete set of data points to start curve fitting. As we will show later in this section, a few data points spread across the exploration space are sufficient in most cases. Once the core availability changes, we predict the next configuration based on the initial data points to minimize the EDP. After each execution step, the EDP value for that period is calculated, which is subsequently used to add or update the reference data points for future curve fittings. The application continues to execute with the best predictable setting. This section discusses the results we obtain under several execution scenarios.

2.4.1 Results without re-threading

We first examine the results with one-dimensional curve fitting (i.e., to adapt to a variation in core availability, we change only the core count). Figure 2.8 shows the EDP values for each of the four benchmarks with different approaches, when using 16 threads throughout the program execution. All the results are normalized with respect to the EDP value when no change is made to the number of cores (i.e., when all the available cores are used); this baseline result is captured by the first bar in each group of bars. In all these plots, we present the results of curve fitting (denoted 1-D fitting) under different numbers of initial data points to start exploration. We
also show the minimum EDP value (the last bar in each group of bars) that can be achieved if the EDP of each step with any setting is known beforehand (so that we can select the best alternative using exhaustive search). This minimum value is also the optimal result that we can possibly achieve using the one-dimensional curve fitting method. The specific selection of initial data points used in these experiments is as follows. When the number of initial data points is 3, 4, 5, 8, and 15, the corresponding sets of core counts with available EDP values are \{2, 8, 16\}, \{2, 8, 16\}, \{2, 8, 12, 16\}, \{2, 4, 6, 8, 10, 12, 14, 16\}, and \{2...16\} respectively; i.e., we start execution with these EDP values ready as our initial data points.

![Figure 2.8](image-url)

**Figure 2.8.** One-dimensional fitting results for FT, MG, LU, and CG. A bar corresponding to a value of \(p\) on the x-axis, gives the normalized EDP value when the number of cores drops from 16 to \(p\). For our curve fitting approach (denoted 1-D fitting), the value within the parentheses indicate the number of initial data points.

As we can see from these plots, when the core availability becomes low (e.g., less than 8 cores for FT and MG or less than 10 cores for LU and CG), there is not much room for improvement. The reason can be found in Figure 2.7, where we see that the EDP values for most execution steps decrease monotonically in certain range of core count as the number of available cores increases. As a result, using all the available cores (i.e., making no change to the number of cores) gives
EDP values very close to the minimum. Similar observation can have been made for LU when the number of available cores is larger than 13 (as the EDP values in Figure 2.7 decrease almost monotonically in this range of core count). In contrast, the EDP savings are significant when the core availability remains relatively high (e.g., larger than 10). We also see that the EDP benefits achieved using curve-fitting guided configurations are not as good as the optimal but much better than the baseline case. Note that, the results with our curve fitting based approach can sometimes be worse than the baseline case. The reason for this is that the actual EDP curves are in general not as smooth as predicted by curve fitting method. Figure 2.9 provides a snapshot of the one-dimensional curve fitting for EDP values in FT. We observe that the curve fitting results closely follow the trend in actual EDP values. However, due to several factors (such as on-chip cache behavior), the EDP value at some point may not be predicted precisely by the curve fitting method (see the data point at 15 cores in Figure 2.9 for an example). Last but not least, we notice that the results may not get better when we start the curve fitting with more initial data points. This is because the initial data points only capture the application behavior in the first execution step and application behavior can change and differ very much as execution progresses (i.e., the initial data points become stale).

In summary, although the specific EDP savings vary with the number of available cores and the number of initial data points we have, our helper thread based method clearly brings benefits in most cases. For example, as shown in Figure 2.8, for the FT application, the EDP savings are 7.4% on average and can go up to 21.7% in the best curve fitting case with 15 initial data points, assuming uniform distribution of the new number of available cores between 5 and 15. Even in the worst curve fitting case with 8 initial data points, the savings we achieve are about 4.8% on average and can be as much as 17.1% for FT.

### 2.4.2 Results with re-threading

Figure 2.10 provides a snapshot of the two-dimensional curve fitting for FT. The circles in the curve fitting surface are the data points used in the triangle-based linear interpolation. Note that the curve fitting results are shifted by 10 to make
both surfaces visible and comparable. Taking this into account, we see that the curve fitting results are reasonably close to the actual EDP values. The average error in this first snapshot shown in Figure 2.10 is around 25% and the error is decreased as more data points are collected during runtime to update the model. Nevertheless, even in this example with few data points, the fitting method still captures the trends in most parts of the surface. The EDP results for the four applications using two-dimensional curve fitting methods are shown in Figure 2.11. All the results are normalized with respect to the optimal value that can be obtained under the one-dimensional curve fitting method, namely, the minimum EDP value if the number of threads does not change. When we consider re-threading, the search space becomes much larger. The two-dimensional curve fitting in these figures starts with 20 data points. These points are chosen randomly from the 120 data points in the lower triangle portion of the space shown in Figure 2.6. Due to the randomness, the plots present the minimum (Min), maximum (Max) and average (Avg) EDP results among 1,000 experiments with random data points in the set. Note that these experiments start with different initial data sets but may end up with some common sequences of selected configurations. Therefore, we do not need to run every simulation completely and the simulation time is greatly saved. For comparison, the minimum EDP value when changing the core count and thread count together is also shown (marked as 2-D optimal in the figures), which represents the optimal result that can be achieved if the EDP value for any configuration at any step is known beforehand.

Our first observation from the results in Figure 2.11 is that changing the number of threads along with the number of cores can reduce the EDP much further. In fact, the minimum EDP values of two-dimensional fitting (2-D optimal) are 31.1%,
Figure 2.11. Two-dimensional fitting results for FT, MG, LU, and CG with 20 initial data points. All bars are normalized to the first bar in each group of bars. The initial data points are randomly selected and results shown here are based on 1,000 experiments.

62.5%, 78.9%, and 89.6% less for FT, MG, LU, and CG, respectively, than the optimal results if we can only change the number of cores. This is due to the much larger search space considered. For instance, when core availability reduces, the number of cores that can be chosen without re-threading is usually less than the current number of threads. This may sometimes exacerbate the load imbalance and waste computing resources. In such cases, changing the number of threads to be the same as the number of cores can possibly give better results. Another observation to note is that two-dimensional fitting also benefits from scenarios where the number of threads is greater than the number of cores. For example, our experimental results indicate that using 4 threads on 2 cores generates lower EDP than using 2 threads on 2 cores for some functions of both FT and MG applications.

The difference between the minimum EDP (2-D optimal) and the best results with our two-dimensional curve fitting approach is minimal in most cases and on average within 5.9% when normalized to our baseline case (instead of normalized
to 1-D optimal in Figure 2.11). Across our experiments with 20 random initial data points, the two-dimensional curve fitting on average reduces the EDP values to be much lower than the optimal of one-dimensional curve fitting (14.9% for FT, 52.5% for MG, 75.4% for LU, and 89.6% for CG), assuming uniform distribution of the new number of available cores from 2 to 15. These savings translate into 26.3%, 59.7%, 77.2% and 91.0% less than the baseline case for FT, MG, LU and CG respectively. Note that the difference between the best case and the worst case for two-dimensional curve fitting is significant sometimes. We found that the higher EDP value in some cases results from a poor distribution of initial data points with few data points around the diagonal in Figure 2.6. The data points on the diagonal represent the scenarios where equal number of cores and threads are running, these points are often close to the optimal settings for most function calls. Therefore, in practice, a careful selection of initial data points can avoid the worse cases and achieve at least the average benefits.

2.4.3 Breaking Down the EDP Reductions

We have seen that the EDP values are significantly reduced using our helper thread based approach. These EDP reductions can come from the reductions in both execution delay and energy consumption. To study the sources of our EDP reductions, we break down the EDP benefits and illustrate the impacts on performance and energy consumption when using our EDP oriented approach.

Figure 2.12 gives an example for the one-dimensional fitting with 4 initial data points (at core counts of 2, 4, 8 and 16) using the FT application. The values of execution delay and energy consumption are separately normalized with respect to the values if no change in core count is made. For comparison purpose, we also studied the exhaustive search approach (denoted 1-D optimal) which assumes to have knowledge of all the possible configurations and obtains the minimum EDP. In each group of bars in Figure 2.12, we show the results for both our one-dimensional curve fitting approach (denoted 1-D fitting) and the exhaustive search approach. In both these approaches, the execution delay is not reduced much and it even slightly increases in some cases. On the other hand, the energy consumption can be significantly reduced, leading to the reduction in EDP. In summary, when we
only change the number of cores and fix the thread count, it is difficult to improve the performance but energy consumption can often become much less.

**Figure 2.12.** Energy and delay breakdown in one-dimensional fitting for FT. All values are normalized with respect to those when no configuration changes are made.

Similarly, Figure 2.13 shows the breakdown of EDP reduction in the FT application for the two-dimensional fitting with 20 randomly selected initial data points. Again, the values of execution delay and energy consumption are separately normalized with respect to the values if no change in core count or thread count is made. In each group of bars, we compare our approach with the exhaustive search based scheme in both one-dimensional (denoted 1-D optimal) and two-dimensional search space (denoted 2-D optimal). As we can see from Figure 2.13, when the number of threads can also be modulated (in addition to the core count), we can improve both performance and energy consumption significantly. Specifically, the execution delay is reduced dramatically because load balance can be improved and thread context switch activities can be minimized with the change of thread count.

Figures 2.14 and 2.15 summarize the results when we break down the energy and delay reductions for MG, LU, and CG in the EDP oriented approach. Similar as in the study for FT, we see that the majority of EDP reductions is due to choosing the optimal number of threads. We also observe that the percentages of reductions in performance delay and energy consumption are similar in Figure 2.15, indicating proper tradeoffs made between performance and energy consumption.
Figure 2.14. Summary of the energy and delay breakdown in one-dimensional fitting with 4 initial data points for MG, LU, and CG. All values are normalized with respect to those when no configuration changes are made.

Figure 2.15. Summary of the energy and delay breakdown in two-dimensional fitting with 20 randomly selected initial data points for MG, LU, and CG. All values are normalized with respect to those when no configuration changes are made.

2.4.4 Results with DVFS

In this set of experiments, we assumed five levels for dynamic voltage/frequency scaling, with the supply voltage ($V_{dd}$) ranging from 0.7V to 1.1V using steps of 0.1V. These numbers are based on the datasheets for several modern cores in similar process technology generation. Under these $V_{dd}$ values, we obtain the scaling factors for frequency, dynamic power and leakage power from HSPICE simulations [32]. We use these scaling factors in calculating the EDP values when the cores are running at different voltage/frequency levels. Note that the curve fitting in this case becomes three-dimensional as we control the number of cores, number of threads, and voltage levels altogether. The experimental results we collected indicate that additional EDP savings are possible over the two-dimensional curve fitting case. Specifically, in comparison to the baseline case, the EDP in FT can be reduced by 46.1% on average and by as much as 66.3% in some cases. Similarly, the EDP saving is 70.5%, 83.4%, and 93.5% on average in MG, LU, and CG; and the savings can go up to 83.3%, 91.2%, and 94.2% in MG, LU, and CG, respectively.

2.4.5 Sensitivity Analysis

As technology scales, it is expected that more processor cores will become available in a CMP [22]. Therefore, it is important to study the effectiveness of our approach
when the number of cores and the number of threads in the system are larger.
We performed experiments with the one-dimensional curve fitting approach on a
system with 24 cores. We observed that our approach is very effective in most cases,
especially when the core availability is not too low. Another set of experiments
investigated the EDP trends under 32 threads running on a CMP system of 16
cores. Recall from Figure 2.8, that little room is there to reduce EDP when the
core availability is very low (e.g., less than half of the thread count). However, we
observed that, with 32 threads, even if the core availability is low, it is still possible
to achieve EDP improvements using our curve fitting based approach. This tells
us that, as the number of threads an application uses increases, there is more room
to make better tradeoffs between performance and energy consumption.

![Figure 2.16. Two-dimensional fitting with the different number of initial data points. We assume that the number of available cores drops from 16 to 14, and the thread count remains at 16.](image)

![Figure 2.17. Two-dimensional fitting with the different number of initial data points. We assume that the number of available cores drops from 16 to 14.](image)

The number of data points used in curve fitting is important in our approach. It
affects the accuracy of the estimations and relates to the goodness of final results.
We performed a sensitivity study that employs different number of initial data
points. Figure 2.16 shows the results for the one-dimensional curve fitting case
with the number of initial data points varying from 3 to 15, when the number of
available cores drops from 16 to 14 and the number of threads is fixed at 16. Figure
2.17 shows the results for two-dimensional curve fitting with the number of initial
data points ranging from 5 to 120, also assuming that the number of available
cores drops from 16 to 14. The EDP values in both figures are normalized with
respect to the results at the smallest number of initial data points. One would
expect better results when more initial data points are used to start curve fitting.
The results with MG in the one-dimensional and two-dimensional cases indicate such a trend clearly. However, the curves become flat beyond a certain number of initial data points. This is because by then the accuracy has already become good enough. The important point is that, for both the one-dimensional curve fitting and two-dimensional curve fitting cases, the number of initial data points needed to reach close-to-best results is small compared to the whole search space. On the other hand, the sensitivity study with FT shows less clear trends. We also observe that in some cases using a larger number of initial data points may even cut the EDP savings. There are several reasons for this behavior. First, as long as some data points close to the optimal settings are chosen, the overall results can be generally good. Second, as mentioned earlier in Section 2.4.1, the initial data points can become stale and the application behavior may not be correctly reflected in curve fitting with more stale data points existing. Third, there might be some unpredictable points in the curve fitting due to unpredicted changes in EDP caused by on-chip cache behavior or other runtime factors.

Although we have only tested class W data set (a relatively small data set) for the NAS benchmarks, the size of input data has no direct influence our helper thread based EDP reduction scheme. As a simple proof of the concept, we carried out a set of experiments for FT and MG with class A data set (which is much larger than class W) and only consider changing the number of cores. The results show similar improvement as reported in Section 2.4.1.

### 2.4.6 Limited Number of Configurations

We have assumed that an execution step of an application can be executed with any arbitrary number of threads after re-threading. This may not always be the case in practice. We now assume that only a few multi-threaded versions exist for the execution steps in our applications. For example, in the FT application, we suppose that only (pre-compiled) 2, 4, 8 and 16 threaded versions of the one-dimensional FFT function (instead of all potential versions for all thread counts from 2 to 16) are available to choose from at runtime. In this case, if the number of the available cores drops from 16 to 14, the version with 8 threads is chosen by our scheme. As a result, the EDP saving we achieve without applying DVFS
is 36.8% over the baseline case. This saving is not as much as the corresponding average EDP saving (42.8%) if any number of threads is allowed, but is still larger than the optimal EDP saving that can be achieved without re-threading (33.2%).

2.4.7 More General Variations

Figure 2.18. One-dimensional fitting for a general core availability variation pattern.

Figure 2.19. Two-dimensional fitting results for a general core availability variation pattern.

Figure 2.20. Difference in the configurations chosen by our two-dimensional fitting scheme and the optimal scheme. Each group of bars indicate the difference in thread count, core count, and EDP value achieved. The EDP value differences are normalized to the optimal values, whereas the differences in thread/core count are in absolute terms.

In all the experiments discussed above, we have assumed that the number of available cores drops from 16 to a lower number, and this core availability change occurs only once. As stated earlier, our helper thread based application adaptation scheme can adapt to the availability variations in the other direction as well, i.e., when the number of cores available to an application increases. In fact, as our scheme is a dynamic one, the number of availability changes that may occur has no effect on its applicability. That is, the number of cores available to an application may change in any direction in any number of times during the course of execution. We now use an example scenario to illustrate the benefits in a more
general case of core availability variation. We assume that the FT application starts execution with 16 cores, and the number of available cores changes in the sequence of 10, 13, 9, 14, 11, 15 at boundaries of the six main iterations of the application. Using our helper thread based adaptation scheme, the EDP can be reduced by about 22% without re-threading, which results from 2.4% reduction in execution delay and 20.2% reduction in energy consumption in comparison to non-adaptation approach. With re-threading, the EDP values are reduced by about 30%, coming from the reductions in execution delay and energy consumption of 15.4% and 22.3% respectively. In addition, the EDP savings can jump to 48.4% when voltage/frequency scaling is also employed. For this availability variation pattern, Figure 2.18 gives the different configurations selected at runtime by three different schemes when the thread count is fixed. The first one is to use all the cores available; the second one is our one-dimensional curve fitting with 4 initial data points (at core counts of 2, 4, 8, 16); and the last one is the optimal one that can be achieved if the EDP values for all potential configurations are known a priori. It can easily be observed that the configurations selected by the one-dimensional curve fitting strategy are close to the optimal selections in most cases. Exceptions may occur when certain unpredictable data points are witnessed as explained in Section 2.4.1. Compared to the optimal adaptation, changes among different configurations are also less when using curve fitting due to the nature of curve fitting method used.

Figure 2.19 gives the results for two-dimensional curve fitting with 20 initial data points. At each step of the FT application, the configuration ([Number of Threads, Number of cores]) chosen by our two-dimensional fitting scheme and the optimal configuration are shown as the points on the corresponding curves. Again, configuration changes happen less frequently when use curve fitting in contrast to the optimal configuration selections. Figure 2.20 presents the difference between the configurations chosen by our two-dimensional fitting scheme and the optimal scheme. We observe that the difference in core count is usually low, but the difference in thread count can be large between the two scenarios. However, even in the cases where the thread count chosen by our approach is largely different from the optimal configuration, the resulting EDP values are very similar in most cases. The EDP differences in some cases are relatively large due to the continuity
of fitting and the lack of data points around the unpredictable optimal configurations. For example, as shown in Figure 2.19, in moving from step 16 to step 18, the configurations chosen by two-dimensional fitting remain at (14, 14), while the optimal configurations in this case would be (15, 15), (12, 12) and (11, 11).

### 2.4.8 Discussion on Overheads and Limitations

Although our helper thread based application adaptation scheme brings significant reductions in EDP as demonstrated by our experimental evaluation presented so far, it is also important to capture and quantify its overheads. Recall that our helper thread collects performance counter statistics, calculates EDP, implements different curve fitting schemes, and changes the number of cores, number of threads and voltage/frequency levels at runtime. We quantified the performance and power overheads of our helper thread and found that its energy overhead (even in the worst case scenario) is about 1% of the total application energy consumption. Since its performance overhead is almost completely hidden in parallel execution, we conclude that the contribution of helper thread overheads to the overall EDP is negligible in practice (note that thread count changes take place in function boundaries in our implementation and this limits the potential overheads significantly).

We also observed during our analysis of the helper thread behavior that most of the overheads it brings are due to implementing the curve fitting strategy. In addition, one can expect the overhead contribution of helper threads to be even lower in the future CMPs where the number of cores and the number of threads will be much higher than the current values we assume in our experiments. To summarize, we can conclude that our helper thread based adaptation approach brings significant EDP benefits even if all the overheads brought by it are accounted for.

It is important to note that the benchmarks we use all have an outermost loop which consists of one or multiple parallel regions. Thus, a different number of threads can be used inside each loop iteration. Codes inside the parallel regions are parallel (i.e., multi-threaded) while those outside are serial (i.e., single-threaded). The granularity that we choose to control the application execution (by selecting the best thread count, core count and DVFS level) is based on execution steps manually defined in Section 2.3.1. In fact, all the execution steps have a parallel
region or multiple parallel regions inside. When considering changing the number of
threads for an execution step, what matters most are the parallel regions inside the
execution step. Consequently, the execution step can be defined at finer granularity
for each parallel region, instead of a function or a group of functions. On the other
extreme, if each iteration in the outermost loop lasts for a short time, a coarser
granularity can be used by combining several iterations as one execution step. A
tradeoff needs to be made between finer granularity and less runtime overheads.

Our study has focused on situations when core availability changes. The ap-
plicability of the proposed adaptation scheme is more broad. Even if there is no
change in core availability as the execution progresses, it might be beneficial from
the EDP perspective to decrease the number of threads/cores or DVFS level. On
the other hand, our approach also has some limitations. For the curve fitting based
EDP prediction to be effective, the same execution step needs to execute repeat-
edly and behave similarly across different outermost iterations. Otherwise, our
approach might perform poorly. Moreover, in some rare cases, if an application
has only one parallel region spanning all the application execution time and no loop
surrounding it, our method cannot be fully applied either because the number of
threads cannot be changed.

2.5 Related Work

This section presents a discussion of the prior work on CMPs, core adaptation,
voltage scaling, application scheduling and other related topics, and compares these
efforts to our work. Chip multiprocessors have been studied in the past from the
perspective of hardware [33, 34, 35] as well as software [36, 37, 38]. In comparison to
these efforts, the work described in this chapter focuses on application adaptation
under varying core availability.

There have been several prior publications on deciding the number of CPUs
to employ in different program phases. Hall and Martonosi [39] pointed out that
compiler-parallelized applications may waste various computational resources in
different program phases. To remedy this, they proposed a mechanism to dynam-
ically adjust the number of CPUs which in turn improved workload performance.
There are two main differences between their study and our work. First, our main
goal is to minimize the energy-delay product, whereas they focused mainly on performance. Second, in their work, the number of CPUs was adjusted to react to program behavior, while our approach aims to adapt the application execution to variations in core availability.

Curtis-Maury et al. [40] built a user-level library framework for online adaptation of multi-threaded codes targeting low-power and high-performance, which changes the processors/threads configuration as the program executes. An online performance prediction model was used to predict performance, power and combined metrics based on phases, and the model was evaluated on a server composed of four processors. In comparison, our work focuses more on adapting application execution to the dynamic core availability. Also, we use a full system CMP simulator and use a power model based on program characteristics, which is not the case in their work. In addition, we employ voltage/frequency scaling.

A few studies have focused on determining the optimal parallelization levels to improve performance at runtime especially for simultaneous multi-threaded (SMT) processors. Zhang et al. [41] proposed an adaptive OpenMP loop scheduler to select the number of threads. Jung et al. [42] used dynamic feedback and runtime decision runs for the same objective. Fedorova et al. [43] implemented an accurate IPC model and demonstrated the performance benefits by running fewer threads than allowed even if more threads are ready to execute. Compared to these work, our work targets at CMPs with one thread context per core instead SMTs and minimizes the EDP. Suleman et al. [44] used a dynamic feedback framework to predict the optimal number of threads based on data-synchronization and off-chip bus bandwidth usage in CMPs, which reduces both execution time and power consumption. In comparison, we consider more parameters of control and predict the EDP directly.

Dynamic Voltage-Frequency Scaling (DVFS) has been widely studied in the past to reduce power consumption of CMPs or other systems. For example, Wu et al. [38] studied the effectiveness of a dynamic compiler-driven voltage scaling scheme. Isci et al. [45] investigated global power management policies to decide per-core DVFS actions with a given power budget. Dhiman and Rosing [46] used online learning method to select the best DVFS settings at any given point in multi-tasking environment. Rajamani et al. [47] implemented runtime power monitoring
and predication modules to tailor DVFS settings to workload demands. Freeh et al. [48] analyzed the energy-delay tradeoff for different applications on a power-scalable cluster with DVFS.

Li et al. [49] explored a multi-dimensional design space for CMPs, which includes the core count and operating voltage/frequencies. However, they did not consider dynamic adaptation at runtime. Li and Martinez [50] discussed the viability of changing the number of concurrent processors/threads at run-time to accommodate changes in the execution environment. Their work studied several heuristics to prune the search space for determining the optimum number of cores to employ and the voltage/frequency levels to use. They assumed at most one application thread is executed on each core, which restricted potential search space. As a result, they did not consider modifying the thread structure of the application. Also, their main target metric was power consumption, whereas we consider energy-delay product, which we believe is a more suitable metric for many data-intensive computing environments.

Job schedulers such as queue-based priority scheduling [51] and gang scheduling [52] have been well-studied in the past to improve the performance and throughput in parallel systems. Snavely and Tullsen [53] defined weighted speedup to measure the performance improvement in a multi-programmed environment and used sampling of the jobs to capture the thread interactions. Our approach is complementary to these scheduling methods for multi-programmed workloads and focuses on how to reduce the energy-delay product for each application in a coordinated manner.

Thread migration in the context of CMP has been studied in the past [54, 55, 56, 57, 58, 59]. Constantinou et al. [54] investigated the performance implications of single thread migration on CMPs. The experimental results they provide show that the performance loss due to activity migration can be kept at minimum when several techniques are used. Thread migration has also been applied in several studies [55, 56] to manage power density. The main difference between most of these prior efforts and ours is that we focus on adapting application behavior to core availability change and target the energy-delay product.

Recently, there have been several efforts on dynamic resource partitioning in the context of CMPs. Guo et al. [60] focused on microarchitecture and software
support in order to provide a guarantee of a certain level of performance. They proposed optimization techniques to improve throughput when applications with diverse requirements exercise the same CMP. Chu et al. [61] proposed a profile-guided method for partitioning memory accesses across distributed data caches. Such a data partitioning reduced stall cycles for memory accesses and achieves overall performance speedup. In comparison, our work focuses on how to utilize the resources with both good performance and low power consumption.

Our work is also different from the classical fault tolerance oriented research [62]. Specifically, our focus is not on the techniques for failure recovery or load balancing. Rather, we try to minimize the EDP at runtime through careful selection of core count, thread count and voltage/frequency levels in a CMP based architecture.

2.6 Conclusions

The main contribution of this chapter is the discussion of an adaptive approach targeting varying core availability in a CMP based environment. The proposed approach makes use of the helper thread concept and reacts to core availability change by selecting the number of cores, number of threads, and voltage/frequency levels to use through curve fitting. We implemented three different variants of our helper thread based approach and performed experiments using four application codes. The experimental results we collected clearly show the success of the proposed approach. For example, when we adapt application execution by changing all control parameters (i.e., thread count, core count, voltage/frequency level), we achieved on average 46.1%, 70.5%, 83.4%, and 91.2% EDP savings in applications FT, MG, LU, and CG, respectively. We also conducted a set of sensitivity experiments where we changed the default values of some of our simulation parameters, and observed that our EDP improvements are consistent across a wide range of experiments.
Dynamic Core Partitioning for Energy Efficiency

In Chapter 2, we have assumed that only one application is running on the CMP at any given time. While there are certain applications that can take advantage of the large number of cores in CMPs, most existing applications have limited parallelism especially if the underlying problem size is kept fixed. Therefore, in the short term, one effective way of utilizing CMP architectures is to execute multiple (potentially multi-threaded) applications at the same time. In this scenario, one critical issue is the management and partitioning of the shared CMP resources such as processor cores and shared on-chip caches. A specific question is to decide how much resource to allocate to each application and how this allocation should be varied over the time. We assume two threads that belong to different applications are not mapped onto the same core at the same time in an attempt to prevent destructive interactions between threads [63]. In this chapter, we focus on the partitioning of available cores across competing applications when multiple multi-threaded applications are simultaneously running on the same CMP.

A large body of recent studies have focused on the dynamic resource partitioning in the context of CMPs, especially targeting performance speedup or quality of service (QoS) [64, 60, 65, 66]. On the other hand, optimization metrics other than performance such as energy consumption are becoming increasingly important in computer system research and practice [67]. To achieve high performance and reduce energy consumption in power-aware high-performance computing, we define
a new metric (which is essentially the average EDP savings when considering all concurrently-executing applications), called the *Weighted Energy-Delay Product Gain* (W-EDPG), for evaluating the behavior of a given workload from an EDP perspective.

The most straightforward method to partition the processor cores is to divide them as evenly as possible across applications. However, this approach might be suboptimal because applications can have different requirements for computing resources. Results in Chapter 2 suggest that the number of cores allocated to an application and the number of threads used to execute it should be dynamically changed to minimize EDP. Also, while it may make sense from a pure performance point of view to increase the number of cores allocated to an application as long as there are additional performance gains, from an EDP perspective this may not always be the best option due to the extra leakage overheads brought by additional cores and caches. As a result, an application may sometimes want to work with fewer cores than available as far as minimizing EDP is concerned.

In this chapter, focusing on a CMP architecture that can be used by multiple, concurrently-executing, multi-threaded applications, we make the following contributions:

- We propose a core partitioning scheme that dynamically divides the available cores across concurrently-running multi-threaded applications to maximize W-EDPG. This scheme, which is based on collaboration between the operating system (OS) and a runtime prediction module, uses curve fitting to predict EDP, considers changes in workload behavior, and partitions the available cores across applications to maximize the value of the W-EDPG metric.

- We evaluate the proposed resource partitioning scheme using different execution scenarios selected out of six multi-threaded applications. The collected experimental results indicate that our approach performs much better, in terms of W-EDPG, than equal core partitioning (i.e., giving each application the same number of cores and keeping it that way throughout the entire execution period). The value of the W-EDPG metric our approach generates is 11.7% on average and as high as 25% in some cases. In addition, we show that the proposed dynamic core partitioning scheme outperforms a hypothetical static scheme (that derives the best static partitioning) as well as an OS based core sharing mechanism.
Our approach performs better than the equal core partitioning scheme because not all the applications have the same resource (core) demands during execution. When an application chooses to use fewer cores than those available to it from an EDP perspective, another application may utilize the unused cores if doing so minimizes its EDP. In this case, the EDP values for both applications could be reduced. In contrast, fixing the number of cores allocated to an application either severely penalizes it (reducing its performance significantly) or allocates to it more resources than necessary (increasing the energy consumption). In both cases, one can expect an increase in the energy-delay product. On the other hand, unlike our scheme, an OS-based scheduling approach may schedule threads from different applications on the same core, and this can lead to significant performance and energy degradations due to destructive interferences among applications [68, 69].

3.1 Target Architecture, Applications, Optimization Metric, and Execution Steps

Architecture. Our target CMP architecture is the same as shown in Section 2.1. We assume that multiple, multi-threaded applications can be mapped onto this CMP at the same time, but two threads that belong to different applications are not mapped onto the same core at the same time. The available cores need to be partitioned among a set of concurrently-running applications at any given time, and it is this dynamic core partitioning our approach tries to optimize.

Applications. We evaluate our approach and alternative schemes using four applications from the NAS Parallel Benchmark Suite 3.2 [25] with the class W input data sets and two applications from PARSEC 2.0 [70] with the class native input data sets. All these applications are parallel (multi-threaded) and implemented using OpenMP. They are chosen to represent different domains. Benchmarks FT, MG, LU, and CG from NAS Parallel Benchmark Suite 3.2 have been introduced in Section 2.3.1. The rest two PARSEC benchmarks are explained as the following:

- Blackscholes: This benchmark applies the Black-Scholes partial differential equation to calculate the prices for a portfolio of European options analytically. It has 100 runs of the price calculation process and we only study the first 15 runs.
- Bodytrack: This benchmark tracks a human body with multiple cameras through a sequence of 260 frames. Due to the long execution time, we only study the first 15 frames.

**W-EDPG.**

The metric that we use in this chapter to compare different resource management schemes is the *weighted energy-delay product gain* (W-EDPG). Let $EDP_i$ be the energy-delay product for the multi-threaded application $i$ when the cores are *equally partitioned* among all the applications in the workload and kept that way throughout the entire execution period. We define *energy-delay product gain* for application $i$ as:

$$EDPG_i = \frac{EDP_i - EDP'_i}{EDP_i},$$

where $EDP'_i$ stands for the EDP of application $i$ when using a different core partitioning scheme. Clearly, we want the value of $EDPG_i$ to be as large as possible. Finally, the *weighted energy-delay product gain* (W-EDPG) of an execution that involves a workload of $n$ applications is defined as:

$$W-EDPG = \frac{EDPG_1 + EDPG_2 + \cdots + EDPG_n}{n}.$$

W-EDPG represents the average improvement in EDP of all concurrently-executing applications (over equal partitioning of cores) in the system. The definition of W-EDPG is similar to that of weighted speedup [53] which has been widely used in studying performance of a workload. The goal of our work is to determine/modulate the number of cores allocated to each application on-the-fly, such that the W-EDPG value is maximized.

**Execution Steps.** To study the dynamic core partitioning across multiple applications, one of the most important issues is to decide the time at which changes in core allocations can take place. As our work is EDP oriented, we need to collect the EDP information when considering new partitionings. However, the EDP of an application cannot be calculated before its execution finishes and thus is hard to control directly. If we consider EDP at the granularity of instructions or time intervals (e.g., EDP per instruction or EDP per OS scheduling quanta), the EDP
for the whole application execution still cannot be inferred as we do not know the total number of instructions or total execution time beforehand.

As in Chapter 2, we rely on identifying execution steps for different applications. Different core partitioning choices are only considered across these execution step boundaries. For instance, we can calculate the EDP for each iteration in the outermost loop and consider a new partitioning for the next loop iteration. Note that, in our target application domain, the number of iterations in outermost loops usually does not change at runtime and thus the partial EDP is useful for estimating and controlling the EDP for the whole application. To define the execution steps, we can also use smaller program structures such as function calls or parallel regions (in OpenMP) to create more opportunities to modulate the core allocations. In this chapter, the same execution steps explained in Section 2.3.1 are used for FT and MG. For the rest four benchmarks (CG, LU, blackscholes, and bodytrack), we define each iteration in the outermost loop as an execution step. Similar definitions of execution steps can be generalized to other applications as well. Several proposals [71, 72] suggest that cooperations between applications and operating system/runtime system can be very beneficial in practice. We envision that execution steps will be specified by programmers or extracted automatically by compilers.

3.2 Empirical Motivation

To demonstrate that uneven partitioning of cores may play a critical role in maximizing W-EDPG, we consider the scenario when two applications (FT and MG) are executed on a 16-core CMP at the same time, assuming one thread per core. Figure 3.1(a) plots the number of cores allocated to FT and MG using exhaustive search to decide the ideal core allocations. In this search process, we try to minimize the EDP for each execution step of individual applications in order to minimize the EDP for the entire execution, which in turn maximizes the value of the overall W-EDPG metric. Each point on the x-axis corresponds to an execution step boundary of either application, as defined earlier in Section 3.1. As the execution step boundaries for different applications are not synchronized, one execution step for an application may span several points in the corresponding curve. Note
also that, MG finishes its execution earlier than FT, thus, its core count drops to zero toward the end. Figure 3.1(b) shows the resulting EDP values over execution steps normalized to the values when using the equal partitioning scheme. The normalized value is set to one when execution finishes. We assume that initially the available cores are divided equally between the two applications, i.e., FT has 8 cores and MG has 8 cores. During execution, an application may release some cores or request more cores in order to minimize its EDP.

We see from Figure 3.1(a) that the ideal number of cores allocated to these two applications changes during the course of execution. More specifically, in some parts of the execution, FT is allocated more cores than MG, whereas in some other parts, it is the other way around. With respect to the equal partitioning of cores throughout the entire execution, this nonuniform, dynamic partitioning brings around 19.1% improvement in the EDP of FT and negligible change in the EDP of MG, resulting in an W-EDPG value of 9.1%. With a closer look at the EDP variations across execution steps depicted in Figure 3.1(b), one can make several observations. First, as expected, the EDP values for both applications are reduced together in certain steps under the nonuniform partitioning. Second, in some execution steps, the EDP of one application is reduced while the other application has the same EDP or increased EDP. The W-EDPG metric for the workload can
still be improved in such cases, provided that the loss in one application is less than the gain in the other application. The reason for the increased EDP in some execution steps is mostly because cores released by one application may have already been occupied by the other when the original owner wants them back. This problem is due to the asynchronous execution step boundaries, and we discuss how to address it in the next section. These results clearly motivate for a core partitioning strategy that can accommodate nonuniform partitions and change the partitions dynamically during the course of execution. In the rest of this chapter, we present and evaluate a practical runtime system/OS based collaborative scheme along this direction.

3.3 Our Dynamic Core Partitioning Scheme

The goal of our work is to determine and modulate the number of cores allocated to each application across execution steps, such that the resulting \( EDP_i \) values lead to maximized \( W-EDPG \). We propose a dynamic scheme, which uses input from a curve fitting based model that predicts the best operating points at runtime. Figure 3.2 shows the main components of our core partitioning approach, consisting of a runtime module (basically a dynamic predictor) and an OS partitioner. The runtime module is invoked whenever an application reaches its execution step boundary. When invoked, this module obtains the number of available cores from the OS partitioner and the performance counter values from the CMP hardware. Using these, it determines the ideal number of cores to be allocated to that application and passes this core count to the OS partitioner. The OS partitioner in turn changes the core allocation for the application and implements re-threading if needed.

**Major Components and Their Interactions.** Our runtime module predicts the ideal number of cores to use in the next execution step for a given application. To do this, it gets the number of available cores for an application from the OS partitioner and uses a curve fitting based model to predict EDP values. The curve fitting based model maintains a list of \([\text{number of cores}, EDP]\) pairs at runtime for the execution steps that perform similar computations. At the end of each execution step, we record the EDP value between the last two execution step
Figure 3.2. The main components of our proposed dynamic core partitioning approach.

boundaries and add it to the set of values collected already. Therefore, the runtime module updates the set of EDP values observed so far during the execution of that application. The EDP calculation uses hardware performance counters to collect performance numbers and an analytical component to compute the required energy values from the performance values and known machine characteristics. This process was explained with more details in Section 2.2. It is important to emphasize that this EDP prediction through curve fitting is a continuous process. As a result, previously-computed EDP data points can be updated before getting “stale”. In any case, our approach uses the most recent EDP values during prediction and consequently dynamically adapts to the execution behavior change between execution steps.

The EDP values during an execution step can be predicted only if we can find execution steps with similar computations that have been executed or profiled. This assumption is reasonable for many applications such as those with loop iterations carrying out similar computations. To start the curve fitting process at the beginning, we may statically profile the application in a small set of data points (core counts) and record the corresponding EDP values over a couple of execution steps. In our experiments to be discussed in the next section, the curve fitting method uses profile data for all applications at core counts of 2, 4, 8, and 16, unless indicated otherwise. The EDP predictions are made for each application independently. For applications with similar execution steps (e.g., FT, LU, CG,
blackscholes, and bodytrack), we use a single model for prediction. For MG where execution steps within a loop iteration have different characteristics, we use a different model for each execution step within a loop iteration but utilize the same model for similar steps across iterations.

After predicting the new number of cores for minimum EDP, the runtime module passes this core count to the OS partitioner. The job of the OS partitioner (which can be integrated with the OS scheduler if desired) is to partition available cores across competing multi-threaded applications at runtime. As stated earlier, the runtime module predicts the ideal core count out of the number of available cores (provided by the OS partitioner). More specifically, if an application is currently using $x$ cores and the OS partitioner also has $y$ idle cores, the new core allocation to this application can only be less than or equal to $x + y$. When the runtime module requests a new allocation, the application can be re-mapped or re-threaded if needed.

As mentioned in Section 3.2, one potential problem is that cores released by one application may have already been allocated to other applications when the first application wants them back. When such a situation arises, a tag associated with that application is set. This tag alerts future core partitioning that one application is predicted to be under-utilizing its equal share with EDP degradation (i.e., based on the curve fitting results, a lower EDP at the current execution step can be achieved with a core count that is more than the currently available cores yet no larger than its equal share of cores). Once a tag is set, when another application that runs with more cores than its equal-share finishes its current execution step, the upper bound for core count of this application is set to be the same as in equal partitioning. The tag will be reset later when the under-utilization problem is solved. In other words, our scheme reclaims from other applications the number of cores that are more than their share in equal partitioning (at execution step boundaries of those applications).

**Example.** To illustrate our core partitioning scheme, Figure 3.3 gives an example that involves four concurrently-executing applications ($A$, $B$, $C$ and $D$) on a 16-core CMP. We assume that initially each application can use up to four cores. The runtime module applies our curve fitting method at time $t_0$ to decide the ideal number of cores to allocate (no more than four) in minimizing the EDP for each
of these applications. At time $t_1$, application $A$ releases two of its allocated cores. As a result, there are four idle cores at time $t_2$ when application $C$ reaches its execution step boundary (two released earlier by application $A$ and two that were not used by application $D$ at the beginning). Therefore, application $C$ can use up to 8 cores in its next execution step (as it already has four allocated cores). However, at time $t_3$, application $D$ cannot use more than two cores even if four cores are predicted (by our runtime module) to be a better operating point. In this case, we set a global tag to alert future core partitioning about the underutilization. At time $t_4$, application $C$ finishes its current execution step with more cores than its equal-share. Its upper bound for core count is set to 4 (i.e., the same as in equal partitioning). By reclaiming these cores, when application $D$ arrives at time $t_5$, it has more available cores to use and the tag can be reset.

![Figure 3.3](image.png)

**Figure 3.3.** An example core partitioning using our scheme when four applications (A, B, C, and D) are executing. Each box represents an execution step. Allocated core counts are specified inside the boxes.

**Limitations.** The implementation discussed above potentially has several limitations. First, the number of cores is chosen to minimize the EDP for the next execution step, instead of minimizing the EDP for the entire execution. However, the latter is difficult to control or to predict directly and our approach is reasonable as these two minimization methods are highly correlated. Second, the implementation is based on a greedy algorithm in the sense that an application could take all the idle cores in the systems even if another application arriving later (finishing its current execution step) also wants more cores and could make better use of them. To address this problem, we add a wish-list table in the runtime module to keep track of the expected gains each application could obtain with more cores. The table includes a list of pairs for each application in the format of [extra number of cores, estimated improvement on EDP savings in percentage] and gets updated
after every execution step. For example, at time \( t_0 \) in Figure 3.3, although we limit the maximum number of cores for each application to be its share in equal partitioning (i.e., four cores for each application), we can estimate the EDP savings each application would gains if more cores are allocated. This information can be used to reserve a number of idle cores for future allocations (instead of giving those cores to the first application that requests them). Consequently, when application C reaches its execution step boundary at time \( t_2 \), the runtime module first considers the reservations from all the applications together and compares how well the idle cores would be utilized under different partitionings. At time \( t_2 \), if application B has core reservations pending, the runtime module decides how many cores to reserve for it after comparing expected savings and the maximum number of cores to allocate for application C may be set to less than 8.

So far, our approach has been focused on how to best utilize the idle cores in the system. However, if no application releases any core in minimizing its own EDP, there would be no optimization opportunities in the previous implementations. A more aggressive approach would be to allow some applications to release their cores voluntarily, if doing so does not significantly affect its own EDP in the next execution step, when there are core reservations with significant expected gains in the runtime module. With the voluntary core release implemented, we could have more opportunities to maximize the value of W-EDPG.

**Algorithm.** Figure 3.4 gives the pseudo code for our dynamic core partitioning scheme. Whenever an application finishes its current execution step, function `RuntimeHandle` is invoked, which calculates the EDP and obtains the number of available cores from the OS partitioner (by calling function `Partition`). After that, a possibly new core count is decided based on the number of available cores and the curve fitting model in the runtime module. This number is then returned to the OS partitioner by calling function `Allocate`, where the partitioner changes the core allocation and links the appropriate static version of the multi-threaded application. Boolean variables `cr_enable` and `vr_enable` are used to enable core reservation and voluntary release respectively. As mentioned earlier, to make a core reservation for an application, the runtime module keeps a list of pairs in the format of [extra number of cores, estimated improvement on EDP savings in percentage]. These lists are used to find the ideal number of cores that maximizes the summation...
\( p \): number of cores;
\( n \): number of applications;
\( \text{flag}[i] \): core reclaim flag for application \( i \);
\( \text{upper}[i] \): upper bound of core count for application \( i \);
\( n[i] \): number of cores allocated to application \( i \);
\( \text{id} \): id of the application that has just finished an execution step;
\( \text{predicted}[s] \): predicted EDP value for the current application when using \( s \) cores;
\( \text{cr-enable} \): boolean value to enable core reservation;
\( \text{vr-enable} \): boolean value to enable voluntary core release;

\# running in the operating system partitioner

\text{Partition}(\text{id})

\text{if} \text{ not initialized}
\quad \text{flag}[\text{id}] = \text{false};
\text{upper} = \left\lceil \frac{p}{n} \right\rceil ;
\text{else}
\quad \text{for each terminated application} \ i
\quad \quad \text{n}[i] = 0 ;
\text{if} \text{ flag}[i] \text{ is true for any application} \ i \text{ and} \ n[\text{id}] > \left\lceil \frac{p}{n} \right\rceil 
\quad \text{upper} = \left\lceil \frac{p}{n} \right\rceil ;
\text{else}
\quad \text{upper} = p - \sum_{i \neq \text{id}} \text{n}[i];
\text{return} \ \text{upper};

\text{Allocate}(\text{id}, s, f)

\text{allocate} \ s \ \text{cores to application} \ \text{id};
\text{link the corresponding static version of the application} \ \text{id};
\text{flag}[\text{id}] = f ;

\# running in the runtime module

\text{RuntimeHandler}(\text{id})

\text{calculate EDP in last execution step based on counters;}
\text{update the curve fitting model based on the new EDP results;}
\text{upper} = \text{Partition}(\text{id});
\text{find} \ s \in [1, \text{upper}] \text{ such that} \ \text{predicted}[\text{s}] \text{ is the minimum;}
\text{if} \ \text{cr-enable}
\quad \text{if} \ s > \left\lceil \frac{p}{n} \right\rceil 
\quad \text{if} \ \text{vr-enable}
\quad \quad \text{compare the losses with voluntary core release and}
\quad \quad \quad \text{the gains associated with core reservations;}
\quad \quad \text{else}
\quad \quad \quad \text{compare the gains associated with core reservations;}
\quad \quad \quad \text{find} \ s \in [1, \text{upper}] \text{ maximizing total gains}
\quad \quad \text{else}
\quad \quad \quad \text{make core reservations with estimated gains;}
\quad \text{if} !\text{vr-enable} \text{ and application} \ \text{id} \text{ is under-utilizing its share}
\quad \quad \quad \text{with EDP degradation}
\quad \quad \quad \ f = \text{true};
\quad \quad \text{else}
\quad \quad \quad \ f = \text{false};
\text{Allocate}(\text{id}, s, f);\)

\text{Figure 3.4.} Pseudo code for the runtime model and the OS partitioner in our dynamic core partitioning scheme.

of gains over all applications. The mechanism is similar when voluntary release is enabled except that the search space is larger. In our experiments, both core reservation and voluntary release are disabled by default unless otherwise indicated.
3.4 Experimental Setup and Results

We implement our approach and perform our experiments using the Simics 3.0 toolset. The experimental setup is similar to what we presented in Section 2.3.2. To estimate the energy-delay product, we utilize the same sets of performance counters and power models. In our simulator, each core runs the Solaris 10 operating system with Sun Studio compilers and tools to support OpenMP. We use the `pset_bind` routine supported by the Solaris to control application-to-core assignment. Table 4.1 gives the major simulation parameters used in this study with their default values. As mentioned earlier, Simics is a full system simulator and the results presented below include all impacts/overheads incurred by our proposed dynamic core partitioning scheme.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>16</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>2GHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64K, 2-way, 2 banks</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64K, 2-way, 2 banks</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>8MB, 16-way, 4 banks</td>
</tr>
<tr>
<td>L1 Access Latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L2 Access Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory Access Latency</td>
<td>260 cycles</td>
</tr>
<tr>
<td>Process Technology</td>
<td>70nm</td>
</tr>
</tbody>
</table>

In our experiments, we evaluate different core partitioning schemes using the workload mixes with two or four applications running simultaneously. With six benchmarks in total, all the workload combinations are evaluated, i.e., 15 two-application workloads and 15 four-application workloads. We use the letter ‘F’, ‘M’, ‘C’, ‘L’, ‘B’ and ‘T’ to denote application FT, MG, CG, LU, blackscholes and bodytrack respectively. For example, ‘FCBT’ stands for running FT, CG, blackscholes and bodytrack together on the same CMP.

![Figure 3.5.](image1.png) Core partitioning for LU and MG with our dynamic partitioning scheme on a 16-core CMP.

![Figure 3.6.](image2.png) W-EDPG for two-application workloads with our dynamic partitioning scheme on a 16-core CMP.
We first investigate the scenario where two applications are running together on a 16-core CMP. Figure 3.5 shows the number of allocated cores by our scheme as program executions progress when LU and MG are executing together. Each point on the x-axis corresponds to an execution step boundary of either application, as defined earlier in Section 3.1. As the execution step boundaries for different applications are not synchronized, one execution step for an application may span several points in the corresponding curve. Note also that, MG finishes its execution earlier than LU, thus, its core count drops to zero toward the end. We see that our scheme successfully modulates the number of cores allocated to these applications at runtime. The resulting W-EDPG value (recall that the W-EDPG metric is defined as the improvement over the equal partitioning case) is 15.0%. This is achieved through reductions in the energy-delay product of individual applications. Compared to the results with equal partitioning, the EDP for LU is reduced by 19.1%, and the EDP for MG is reduced by 10.9%. Our core modulation does not bring much performance (execution latency) overhead. Instead, it results in speedup in some cases because re-threading enables an application to use cores released by others after they finish executions. For example, following the dynamic partitionings shown in Figure 3.5, the execution time is decreased by 20.9% for LU and 12.9% for MG, both with respect to equal partitioning. Note that sometimes the minimum EDP values are achieved using fewer cores than available. As an example, in Figure 3.5, LU uses only four cores in some execution steps, even though eight cores are available. Figure 3.6 illustrates the W-EDPG values for all the two-application workloads using the proposed dynamic core partitioning scheme. The arithmetic, harmonic and geometric means (among all workloads) are also plotted in the figure. Each label on x-axis indicates the workload mix. The average value of W-EDPG is around 8%. We observe minimal negative W-EDPG values (less than -0.5%) in some cases. This is due to the overheads of our approach when our method selects the same or extremely similar core partitionings as equal partitioning during the execution.

Figure 3.7 gives the W-EDPG values for all the four-application workloads when using our dynamic core partitioning. We see that the geometric mean of the improvements is 14.0%. Compared to the two-application workloads, the W-EDPG is higher when more applications are running concurrently. There are
two main reasons behind this. First, with more applications, more optimization opportunities exist. Core releases or requests from different applications create a larger space to explore. Second, when one application finishes its execution, more applications can take advantage of the idle cores if EDP can be reduced by doing so. For example, Figure 3.8 gives the number of allocated cores when scheduling FT, MG, CG, and LU together on a 16-core CMP. As a result of the shown partitionings, the EDP results for FT, CG and LU are reduced by 15.4%, 24.6% and 14.2% respectively when compared to the equal partitioning scheme. However, the EDP of MG is increased by 14.1%. This is because there are fewer available cores than its share in equal partitioning in one of the longest execution steps in MG. Nevertheless, the overall W-EDPG value of the workload is 10.0% because the EDP savings on other applications are significant. When we look at the performance results, we see that FT, CG and LU achieve speedups of 14.3%, 21.7% and 19.5%, respectively, but MG is slowed down by 14.6%.

We also carried out the same set of experiments with the core reservation scheme enabled (see Section 3.3). Figure 3.9 depicts the core allocations when running FT, MG, CG and LU on a 16-core CMP in this case. The resulting W-EDPG is 9.8%, with EDP reductions of 10.2%, 25.5% and 7.7% for FT, CG and LU, respectively, and an EDP increase of 4.3% for MG. When we further enable the voluntary release, the results were similar. Comparing Figures 3.8 and 3.9, we can make two observations. First, with the core reservation enabled, more applications can take advantage of the idle cores and the variation of benefits achieved by different applications gets reduced. Second, the W-EDPG may get lower with the core reservation. This is because the reserved cores may be left idle for a long period before an application can actually seize them. We need a
prediction model for the execution step boundaries in order to take into account the waiting time for reserved cores to be utilized. Note that such prediction is much more difficult than predicting the best core count for an execution step. Our curve fitting based prediction works well for the latter because we can tolerate prediction errors as long as the order of EDP values is preserved. In this case, the prediction of execution step boundaries needs much more accuracy and is beyond the scope of this chapter.

Figure 3.9. Core partitioning for FT, MG, CG and LU with core reservation enabled in our dynamic partitioning scheme on a 16-core CMP.

Although our approach brings significant improvements in W-EDPG as demonstrated by our experimental evaluation, it is also important to capture the performance and energy overheads incurred by curve fitting. We quantified these overheads and found that both the performance overhead and the energy overhead are less than 1% of the total cost for an entire execution of the applications. The EDP-overheads incurred by our approach vary between 0.2% and 0.5%. Therefore, the overheads to the overall EDP is negligible in practice. As mentioned earlier, the presented results include all the overheads incurred by our partitioning scheme.

3.4.1 Alternative Schedulings

The results above are based on the assumption of one thread per core. Therefore, the application gets re-threaded if the number of cores allocated to it changes. We also have an alternate implementation of our scheme where we change only the number of cores, not the number of threads. This alternative is certainly easier to implement compared to the one thread per core scenario, as it does not require re-threading and just asks the underlying OS to schedule the same number of threads over a new core count. In this case, we allow multiple threads of the same application to share/compete for the resources on the same core. We studied this
alternative for all the multi-programmed workloads, and found that the geometric mean of W-EDPG is 15.5% for two-application workloads and 17.2% for four-application workloads. Due to space concerns, the detailed results for individual workloads are omitted. As an example, when FT and MG are both running fixed 16 threads, the results showed that, compared to equal partitioning, the EDP for MG is reduced by 30.7%, whereas the EDP for FT remains the same. These savings correspond to a W-EDPG value of 15.3%. Note that, in this case, the equal partition also uses fixed 16 threads (and only the cores are partitioned equally), which is different from the equal partitioning used so far. Thus, the results are not directly comparable to those discussed earlier. Therefore, we can conclude that our approach performs well with both the one thread per core assumption and an alternate scheduling scheme that schedules a fixed number of application threads on a given set of cores (using the underlying OS scheduler).

We also performed experiments with a pure OS based approach, where all applications are executed in parallel on all available cores without any core partitioning across applications. We observed that both execution latency and energy consumption in this case were dramatically higher than both our scheme and the equal partitioning case. Specifically, the average performance slowdown for all four applications was about 2X and the total energy consumption almost tripled, both with respect to the equal partitioning case. These are due to the destructive interactions between applications and extra scheduling overheads involved in context switching. We expect this to be the general trend when multiple, multi-threaded applications share the same set of cores in an unrestricted fashion.

While the results presented so far clearly show that our proposed dynamic partitioning scheme performs much better than equal partitioning of processor cores across competing applications, it is not clear whether equal-partitioning is really the best static partitioning for the cases tested. To check this, we performed an additional set of experiments where we exhaustively tried all static partitionings of cores across competing applications. For example, in the case of the workload that consists of FT and MG, we tried the options that give 2 cores to FT and 14 cores to MG, 3 cores to FT and 13 cores to MG, and so on. We observed from these experiments that such fixed static nonuniform partitionings generate in general worse results than the equal partitioning of cores as far as the W-EDPG metric is
concerned. This is because a specific nonuniform partitioning usually only benefits a couple of execution steps while hurting the remaining execution. In fact, the W-EDPG values under such nonuniform static partitioning were always negative and ranged from -1.1% to -131.7%, underlining the importance of dynamically adapting core partitioning at runtime.

3.4.2 Higher Number of Cores

As technology scales, more processor cores will become available in a CMP [22]. Therefore, it is important to study the effectiveness of our approach when the number of cores is larger. To study this, we performed experiments with a 32-core CMP. The simulation parameters in Table 4.1 remain the same except that the size of shared on-chip L2 cache is increased to 16MB. Figure 3.10 plots the W-EDPG values for all four-application workloads when using our dynamic partitioning scheme on 32 cores. We achieve an average W-EDPG value of 12.8%. Note that the baseline is running each application with 8 threads and 8 cores in this case.

![Figure 3.10. W-EDPG for four-application workloads with our dynamic partitioning scheme on a 32-core CMP.](image)

So far, we have assumed that the application can be re-threaded to any number of threads. For the parallel applications implemented in OpenMP, the OpenMP runtime libraries support dynamically changing the number of threads on a parallel region basis during execution. Setting thread count for an execution step eventually translates into setting thread count for parallel regions within the execution step. For applications not implemented in OpenMP, static versioning can be used by preparing different versions at compile time. However, preparing all possible versions may not be feasible if the number of cores in the CMP is very large and the range for possible thread counts is also large. To address this issue, we can
have static versions ready for a relatively small set of pre-determined core counts. In fact, one can easily see from our results for the specific workload shown in Figure 3.5 that the same improvement can be achieved if we only prepare versions of the applications at core count 4, 8, 12, and 16 (instead of all core counts between 2 and 16).

### 3.4.3 Idle Core Repartitioning

In the results above with the equal partitioning scheme, we have assumed that the processor cores are turned off to save power when the application running on them finishes execution. An alternate approach would be to distribute those cores among the remaining applications. Note that this requires an implementation of re-threading on top of our equal partitioning baseline as we assume one thread per core. Adding a couple of extra cores can improve the performance significantly especially when the number of cores already allocated to each application is small. This may lead to savings in EDP if the increase in power consumption can be paid off by the additional performance improvement. On the other hand, the performance improvement could be marginal sometimes especially when the number of cores already allocated to each application is high.

We tested this approach with our multi-programmed workloads on the simulated 16-core CMP. The W-EDPG results for a set of representative workloads are shown in Figure 3.11. Note that we included several three-application workloads as well. For comparison, the results for our dynamic partitioning scheme are reproduced in this plot. Each label on x-axis indicates the workload mix. We see from Figure 3.11 that the modified equal partitioning performs worse than the baseline equal partitioning in some cases, especially in two-application and three-application workloads. In most cases, the W-EDPG is improved with idle core repartitioning, though the improvement is less than that achieved by our dynamic partitioning scheme in most tested scenarios. We also observed that the modified equal partitioning can outperform our approach in certain scenarios. For example, when the two approaches both end up with the same core partitionings, the overheads caused by our curve fitting based prediction cause a slightly lower W-EDPG value.
3.4.4 Different Executing Patterns

So far in this chapter, we have assumed that multiple applications start their executions together. In other scenarios, applications may come in and leave the system at any time. To capture this, we carried out another set of experiments with four-application workloads. Figure 3.12 gives a sample core partitioning when using our dynamic scheme with core reservation. In this scenario, FT and MG start together while CG and LU arrive later. Important application arrival/exit points are marked in the figure. As can be seen from this plot, our approach is able to handle the dynamic application start/exit patterns as well.

3.4.5 More on Executing Steps

As discussed earlier in Section 3.1, the execution steps can be defined by an average programmer or provided by automatic program analyzer. The only requirement is to identify similar computations that take place within the application such as loop iterations. One can adopt simple strategies to define execution steps even without understanding program phases or specific computations. For example, the execution step we choose is simply an iteration of the outermost loop in CG, LU, blackscholes and bodytrack; and execution steps in MG are obtained by dividing the function calls within the outermost loop into several function groups. We can also take each parallel region as an execution step in the OpenMP codes (which is very easy to identify for a compiler), though this may lead to too many execution steps and require a large number of prediction models. The different ways to define execution steps can affect the amount of benefits that can be achieved through
our approach. For example, there is a tradeoff between fine grain and coarse grain execution steps. If there are too many execution steps, we may incur too much overhead at runtime. At the other extreme, one may not have too many optimization opportunities with the small number of execution steps. For example, if we define an execution step as three loop iterations in CG and LU benchmarks and schedule them together on our 16-core CMP, the value of W-EDPG drops from 13.6% to 5.2%. Also, if a execution step runs for a very long time, it may dominate the use of resources and cause fairness problems. To improve fairness, techniques such as varying time slices for different applications [68] can be used at a higher level in the operating system.

3.4.6 Time Sharing vs Space Sharing

In this study, we have implicitly assumed that the number of applications being executed simultaneously is less than the number of cores on a CMP. As a result, we adopted a partitioning across the spatial domain only, without allowing a core to be shared by threads that belong to the different applications. This assumption is reasonable as the number of cores is projected to keep increasing in future [22]. On the other hand, our proposed dynamic core partitioning scheme can also be tailored to work with time-sharing based core management schemes, though the implementation would be much more complex and is beyond the scope of this work. Time-sharing based scheduling usually operates at the granularity of fixed time slice (quantum), whereas our core partitioning scheme introduces a different granularity (of execution step). This brings up several issues unique to our scheme if we want to use it along with time-sharing of cores. First, the calculations of energy and delay have to reflect context-switching, which can be done by recording more performance and energy values for different applications during every time slice and aggregating them for the complete execution step. Second, when an execution step spans several time slices, the number of cores available to it may vary during different time slices. Inside an execution step, if more cores are available than selected in the previous time slice, we can leave them idle and postpone the possible utilization to the next execution step; if the available cores are not enough in a time slice for one application, it would not be
scheduled by the time-sharing operating system to run in that time slice, assuming
the threads of a parallel applications are enforced to get scheduled together using
techniques similar to gang scheduling [52].

The last issue we want to discuss is the optimization metric used in this work. While in this study we used an EDP based metric called W-EDPG, our dy-
namic core partitioning approach can be made to work with other potential global
(workload-wide) metrics as well. For example, several recent papers [67, 60, 53]
proposed different performance centric optimization metrics, which can also be
accommodated within our curve fitting based scheme.

3.5 Related Work

Processor allocation policies have been widely studied in the context of large scale
shared-memory [73, 74, 75] and message-passing [76] multiprocessors. Hall and
Martonosi [39] discovered that compiler-parallelized applications may waste vari-
ous computational resources in different program phases. Tucker and Gupta [74]
proposed a method to control the number of processes associated with an appli-
cation dynamically to improve performance. Compared to these studies, our work
focuses on CMPs and targets at energy efficiency.

Curtis-Maury et al [77, 40] built a user-level library framework for online adap-
tation of multi-threaded codes targeting low-power and high-performance, which
changes the processors/threads configuration as the program executes. Their later
work [78] generalized the prediction models to multiple dimensions and evaluated
additional benefits brought by DVFS. Our earlier work [79] proposed a helper-
thread based scheme for CMPs that predicts the ideal number of cores, number of
threads, and voltage/frequency levels for the single application execution scenario.
In this chapter, we focus on multi-programmed workloads on a CMP and decide the
best core partitioning across concurrently-executing multi-threaded applications.

Job scheduling in multi-processor systems has been well-studied in the past
[52, 51]. Snively and Tullsen [53] demonstrated that the performance of a multi-
threaded processor is sensitive to the set of co-scheduled jobs. DeVuyst et al [80]
studied an adaptive thread scheduling policy which explores both balanced and
unbalanced schedules for multiple single-threaded applications in CMP. Baskaran
et al [81] proposed an approach to improve load balance by dynamic scheduling parallel tiles on the processor cores based on run-time extraction of data dependencies.

A lot of research interests have focused on dynamic resource partitioning for CMPs in recent years. Nesbit et al [72] identified the inadequacy of current resource management mechanisms for multicore systems and advocated a hardware/software interface based on the virtual private machine abstraction. Lin et al [66] designed an OS-based cache partitioning mechanism and evaluated several representative cache partitioning policies with different optimization objectives. Fedorova et al [68] described a software solution to improve performance isolation on CMPs by changing the thread timeslice at the OS level based on the contents of the hardware performance counters. Guo et al [60] focused on microarchitecture and software support in order to provide a guarantee of a certain level of performance. Bitirgen et al [64] built a framework for coordinated resource management considering L2 cache, off-chip bandwidth and total power budget. In comparison, we focus on a metric for exploring the tradeoffs between performance and energy consumption and investigated the benefits by exposing application-specific characteristics to the OS and runtime system. Several other studies discussed static [82, 83] or dynamic [84, 85] resource partition in SMT. In comparison, we assume that different applications do not share cores at the same time and our main focus in this chapter is not on the thread interactions on a given core.

3.6 Conclusions

The main contribution of this chapter is a dynamic, nonuniform core partitioning scheme for multiple multi-threaded applications that use the same CMP concurrently. Our approach is a collaborative effort between the OS and a runtime system module. It uses a curve fitting based strategy to determine, for each application, the ideal number of cores to use to maximize a global (inter-application) metric called the weighted energy-delay product gain (W-EDPG). To test the effectiveness of our approach, we implemented it and compared against alternative schemes under various execution scenarios that involve different numbers of concurrently-running applications and different numbers of cores.
The results collected through our experiments indicate that our approach improves the W-EDPG metric significantly (e.g., 14.0% on average over the equal partitioning scheme for four-application workloads on a 16-core CMP. In addition, our experiments show that the proposed approach generates much better results than traditional OS based core sharing as well as any static core partitioning.
Chapter 4

QoS Aware Dynamic Time-Slice Tuning

The ability to manage quality of service (QoS) and to provide service differentiation has been very important in a wide range of computing environments, especially in computer networking [86, 87, 10, 88, 89] and real-time systems [90, 91, 92]. The primary goal behind QoS based studies is to provide different priority levels to different applications, or to guarantee a certain level of performance. For example, a network routing strategy may provide several priority levels such that different types of packets can be dispatched with proper priority levels to satisfy the latency/throughput requirements. Ensuring the quality of service is critical for many applications. For instance, real-time streaming multimedia applications usually require certain number of frames generated per second.

With multicore architectures dominating the landscape of microprocessor market, the lack of quality assurance on such processors has become a major concern recently. This is because concurrently-executing applications can compete arbitrarily on shared resources such as cores, shared caches, and off-chip memory bandwidth. As an example, cache trashing could happen when data used by an application (its working set) in the last level cache gets evicted by data accesses issued by other applications. Similar issues brought by unmanaged resource sharing have drawn a lot of research interests to study the QoS on general purpose applications and processors [60, 93, 94, 95]. As the number of cores per chip is expected to increase, future computing environments will host increasingly larger number of
applications simultaneously to fully utilize on-chip resources. Managing the QoS and providing service differentiation are clearly critical tasks in this context.

Many recent studies focused on the management of shared resources especially the shared last-level cache. Various hardware cache partitioning/reservation schemes were proposed to provide certain level of control for the quality of service in multi-application workloads [93, 65, 96, 95]. Software based techniques such as page coloring [97, 66] and contention-aware scheduling [98] have also been explored. Most of these previous studies have assumed applications do not share processor cores, i.e., different applications are pinned into different cores. While such simplifications are important to study the sharing of other on-chip resources (e.g., caches) in an isolated manner, they also hide another dimension to optimize. In modern operating systems, multiple applications share processor cores and take turns to execute. Each application typically runs for a while before its CPU time-slice (allocated quantum) expires or the execution is blocked due to I/O operations etc. This gives users the transparent feeling of simultaneously executions even on single-core machines and is usually referred as time sharing. Time sharing [99, 51] has been commonly implemented for a long time on both serial and parallel systems.

In most operating systems, the actual CPU time-slice for a process is based on a base time-slice and some dynamic adjustments. The base time-slice is the same for processes within the same priority level. Minor runtime adjustment to the base time-slice is possible as per the dynamic characteristics such as average sleep times. On the application side, the process priority and associated time-slice assigned by the operating system (OS) scheduler can usually affect its performance. The performance metric we focus in this work is the IPC (instruction per cycle). When multiple applications compete for the CPU cycles in a round-robin manner, longer CPU time-slice often results in larger fraction of CPU execution and leads to performance speedup for an individual application. Note that the significance of performance impacts due to different time-slices can vary from application to application. CPU-intensive applications are more likely to benefit from a large time-slice as opposed to I/O-intensive applications.

In this chapter, we explore the opportunities by dynamically tuning CPU time-slices in order to satisfy the QoS requirements of concurrently-executing appli-
cations that share the same CPU. From an OS perspective, selecting a proper length of the CPU time-slice is often a tradeoff between providing interactive responses and avoiding excessive context switch overheads. Usually, each process is assigned a fixed base time-slice. While fixing the base time-slice for each application makes implementation easy, it also limits potential service differentiation. Even in systems that allow service differentiation or dynamic adjustment of time-slice/priorities, no QoS is guaranteed.

To control QoS in the context of time-sharing workloads, we propose a dynamic CPU time-slice tuning framework using formal feedback control. Feedback control theory provides an efficient way to measure and compensate system disturbances while regulating the values for system parameters [100]. We use fraction controllers to translate the specified IPC values into the CPU fractions that need to be allocated. A multi-input and multi-output controller then takes into all these fractions and decides the required time-slices and corresponding nice values to enforce the CPU fractions. Performance characteristics of each application are measured dynamically at runtime and fed back to complete the control loop. We evaluate our approach on a full-system simulator with a set of benchmarks. The experimental results show that the proposed feedback control based approach can help modulate the usage of CPU shares to satisfy the QoS requirements of concurrently-executing applications in a best-effort matter. We also discuss how to extend our approach to multicore machines.

4.1 Background and Motivation

Most computing systems have a large number of processes running concurrently. An OS scheduler usually classifies the processes to be scheduled into several categories, including interactive, real-time, and batch processes. Interactive and real-time processes have their own requirements for response time and are usually given higher priority. Their scheduling methods have been well studied in the literature [101, 102]. In this chapter, we focus on general-purpose batch processes that have no requirements for response time but have certain other QoS/performance requirements. Different metrics have been proposed and used in the QoS related studies. For example, in the context of cache/memory management, Iyer et al. [93]
discussed three kinds of metrics, including resource usage, resource performance and overall performance. It is usually much easier for users to provide a requirement on the overall performance, instead of the usage/performance information of physical resources directly. Therefore, most recent work on QoS enforcement [93, 64, 103, 65] rely on overall performance metrics to define the target. Similar to these efforts, we use IPC (instruction per cycle) in this chapter as the QoS metric to specify the desired service level (performance). We assume that the users would specify a target IPC value for each concurrently-executing application that has QoS requirements.

![Figure 4.1. Illustration of IPC variations and actual CPU fractions when using different nice values: (a)&(d) default nice values; (b)&(e) bzip2 with nice value -5; (c)&(f) bzip2 with nice value 5.](image)

Processes that simultaneously execute on a time-sharing OS take turns to utilize the CPU. Typically, each process is allowed to run for a period of time (also known as quantum) before a context switch takes place. In this chapter, we use the terms CPU time-slice and quantum interchangeably. Although different OSs normally employ different scheduling strategies, most of them have a base quantum associated with each process based on its static priority. For example, in Linux 2.6, each process has a nice value related to its priority and the base CPU time-slice (in ms) can be calculated as the following [99]:

\[
\text{time-slice}_{\text{base}} = \begin{cases} 
(20 - \text{nice}) \times 20 \text{ ms}, & \text{if nice} < 0; \\
(20 - \text{nice}) \times 5 \text{ ms}, & \text{if nice} \geq 0.
\end{cases}
\]
The base CPU time-slice is dynamically adjusted by the OS scheduler to reflect certain runtime characteristics such as executing time before blocking. The nice value of a process can be changed at runtime with system calls. With different nice values and different base CPU time-slices, an application can execute at different paces. For example, Figure 4.1 plots the IPC values and CPU fractions of bzip2 and gcc (both from SPEC2006 [104]) on a simulated CPU when we apply different nice values (simulation details are explained in Section 4.3). The x-axis in these figures corresponds to different epoches of executions. Each epoch includes the execution of 10 million instructions and this instruction count includes all the processes in the OS. Figure 4.1(a)\&(d) corresponds to the scenario using default nice values for both applications (which are zeros). The nice value for bzip2 is statically set to -5 and 5 in scenarios depicted in Figure 4.1(b)\&(e) and Figure 4.1(c)\&(f), respectively. In these experiments, gcc uses default nice value of 0 and no changes are made for other system processes. One can see that the IPC values vary when we choose different nice values for our applications. With a lower nice value, the fraction of the CPU time spent on bzip2 increases which in turn improves its IPC and decreases the IPC value of gcc. On the other hand, with a higher nice value, bzip2 gets less CPU fraction and the IPC for gcc increases. Overall, we see that controlling the nice values can exercise significant impact on performance (IPC value achieved).

Feedback control theory [102] is an established method widely used in many engineering disciplines. The controller starts with a reference input that specifies the target goal for the system. Different actions are taken based on the output of the system to match the reference input. Consider as an example the problem of controlling room temperature using an air conditioner. In this case, the desired room temperature is the reference input, and the measured room temperature is the output. Based on the difference between the desired and measured temperature values (usually called error), heating/cooling components are turned on and different fan speeds can be chosen. In our problem domain, the reference input would be the target IPC values that users specify for their applications. Figure 4.2 depicts the framework of our feedback control loop. We will explain the details of different components in the next section. The advantages of using such a control theory based approach include tolerance to model inaccuracy, resilience to
parameter and environmental variations, and robustness in tracking the reference input.

4.2 Dynamic Time-slice Tuning Controller

In this section, we propose a feedback control based framework to satisfy the QoS requirements by dynamically changing CPU time-slices given to concurrently-running applications. As shown in Figure 4.2, each application has a fraction controller, which translates the specified IPC target into the fraction of CPU time that needs to be allocated. These desired CPU fractions may not be satisfiable, the nice controller takes into all these fractions for possible reassignment and then decides the required time-slices and corresponding nice values to enforce those fractions. The actual IPC value and CPU fraction of each application are measured dynamically at runtime and fed back to the control loop.

4.2.1 Fraction Controller

Let us assume there are $N$ applications running with their specified QoS targets. Besides these applications, other OS related processes or service daemons usually exist and share available on-chip resources. Our focus is on controlling the $N$ applications with QoS specified. Specifically, we want to track the reference IPC values provided by users for these $N$ applications.

As mentioned earlier, the IPC value is related to the fraction of time that an application occupies CPU in time-sharing systems. We use the variable $Frac$ to denote the fraction of CPU time that is spent on each application. $Frac_i$ and $Frac_{rest}$ represent the fraction of application $i$ and that of all the processes without...
any QoS target. It is trivial to observe that:

$$\sum_{i=1}^{N} Frac_i + Frac_{rest} = 1.$$  \hfill (4.2)

If we could double the CPU fraction for application $i(Frac_i)$ and assume the application behavior does not change, its IPC would be doubled as well. This is because the number of instructions issued by the application would double within a fixed period of time. Figure 4.3 displays the correlation coefficient between IPC values and CPU fractions (over the past five epoches) at different points when four SPEC benchmarks execute together on one processor (benchmarks to be explained in Section 4.3). We see that these two variables are closely correlated most of the time. In fact, the correlation coefficient is more than 0.95 for over 95% of time. On the other hand, there are certain points that they are not much correlated due to the disturbance from the system. We have similar observations with other workloads in our experiments. These observations can be used to determine the desired CPU fraction (also called CPU share) to satisfy the QoS requirement, based on the relationship below:

$$\frac{IPC_{ref,i}}{IPC_i} = \frac{Frac_i'}{Frac_i}.$$ \hfill (4.3)

Here, $IPC_i$ and $Frac_i$ are the current IPC value and CPU fraction for application $i$, respectively. Given the specified QoS target ($IPC_{ref,i}$), one can obtain the desired CPU fraction for each application ($Frac_i'$). To account for changes in application behavior and execution environment, one can choose appropriate intervals such that characteristics between consecutive intervals do not vary widely. The default interval in our study is 10 million instructions. Note that this instruction count includes instructions executed for all the processes. As a result, when we consider workload with more applications, the progress that each application made within the interval is actually smaller. We tested different interval lengths and found out that intervals of 10 million instruction work reasonably well. The fraction controller also tracks the correlation coefficient to identify the presence of significant disturbance and to avoid apply Equation 4.3 in such cases. When the correlation coefficient over the past five epoches is lower than 90%, the desired CPU fraction is kept the same as the measured CPU fraction.
4.2.2 Nice Controller

The desired CPU fractions produced by the individual fraction controllers may not be satisfiable when considering all the applications with QoS targets together. For example, the sum of desired fraction could be easily above 1 if the reference IPC values are set too high. To handle multiple CPU fractions, the nice controller negotiates among the desired fractions and then changes the nice values for certain processes. Note that the nice values for different applications are dependent on each other in order to enforce the CPU fractions. This is why our nice controller is in the multi-input and multi-output form. As the base time-slice is a function of the nice value (refer to Equation (4.1)), the problem of selecting proper nice values is essentially to derive the ideal base time-slices for applications with QoS targets.

• Fraction and Time-slice Model. We use $q_i$ to denote the base time-slice (quantum) that is chosen for application $i$. According to Equation (4.1) given in Section 4.1, the maximum and minimum base time-slice in Linux are 800ms and 5ms, respectively. Intuitively, a process can utilize larger fraction of CPU time when it is assigned a larger quantum. This is because the process can potentially run longer without context switch. However, I/O-intensive processes are usually blocked before their time-slice expires. Even for CPU-intensive processes, as there are other high-priority system threads running simultaneously, they may not be able to use up their time-slices before switched out. Consequently, the CPU fraction often becomes saturated when $q_i$ is increased to certain extent. For example, Figure 4.4 provides part of the CPU fraction curves when we run bzip2 and milc together. Note that each curve is independent and we only change the quantum
for bzip2 or milc in separate experiments. The CPU fraction of bzip2 is much higher than that of milc. This is because bzip2 is CPU-intensive and milc is I/O-intensive. Both curves increase initially and become saturated in the end. We observed the same trend for all our benchmarks and thus use the following saturating non-linear function to model the relationship between the base time-slice and the CPU fraction:

\[ \text{Frac}_i = \text{Frac}_{i,\text{max}} (1 - e^{\alpha_i \times q_i} \). \]  

(4.4)

\( \text{Frac}_{i,\text{max}} \) and \( \alpha_i \) are constant coefficients for application \( i \), which can be calculated either off-line or at runtime. Note also that the differentiation between actual CPU fractions and base time-slices for each application as \( \frac{\text{Frac}_{i,t} - \text{Frac}_{i,t-1}}{q_i,t - q_{i,t-1}} \) can be used to identify I/O-intensive applications or phases and to infer how effectively base time-slices can affect the CPU fractions. Here, \( q_{i,t} \) and \( \text{Frac}_{i,t} \) are base time-slice and measured CPU fraction for application \( i \) during epoch \( t \). Similarly, variables \( q_{i,t-1} \) and \( \text{Frac}_{i,t-1} \) are used for epoch \( t - 1 \).

**Fraction Reassignment.** The desired CPU fractions need to be reassigned to satisfy different constraints such as Equation (4.2). Before the reassignment, we need to estimate the total CPU fraction that could be allocated to the processes with QoS targets. For example, comparing the CPU fractions for three different scenarios depicted in Figure 4.1, one can see that the sum of CPU fractions for bzip2 and gcc is almost the same no matter what nice values are selected. There are two major reasons behind this observation. First, these applications are CPU-intensive so there are little system idle time which could otherwise add noise to the fractions. Second, system processes that are real-time or interactive processes remain their original executing paces so their fractions do not change much. Figure 4.5 plots the CPU fractions when executing two I/O-intensive applications (cactus and milc in SPEC 2006) together. As expected, they only take a small fraction of the CPU time and a large fraction of CPU time is spent in idle mode. In the nutshell, the CPU fraction of system services does not change much across execution epoches, whereas the fraction of CPU idle time incurred by I/O-intensive processes could be reallocated to other processes. Therefore, we can measure the actual CPU fractions at runtime for different processes and infer the limit for the next execution epoch.
Once we decide the limit for the total CPU fraction for processes with QoS targets, we can proceed with the reassignment, which maps $Frac'_i$ to $Frac''_i$. If all the applications would ask for a larger value for $Frac'_i$, there could be a shortage of CPU share. On the opposite side, there might be extra CPU shares available if requests are all for moderate or small fractions. To make full use of the available CPU resource, we need to handle both over-subscription (sum of the required fractions is greater than the limit) and under-subscription (sum of fractions is less than the limit) cases. A simple way to handle this is to linearly scale $Frac'_i$ ($i = 1...N$) but it has the drawback that all the processes would be equally penalized if one of the desired CPU fraction is too high. For example, let us suppose $A$ and $B$ take 40% of CPU time each during the past epoch and they need 30% and 60% respectively for the next epoch. If we scale the sum, $A$ would get $30\% \times (40\% + 40\%)/(30\% + 60\%) = 26.7\%$ and $B$ gets the rest 53.3%. In this case, $A$ would miss the desired CPU fraction due to a demanding request from the other process even though the desired fraction of $A$ is satisfiable.

Our goal in the fraction reassignment is to satisfy as many requests as possible while providing fairness. We first compare the difference between the CPU fraction requested and the current fraction for each application $i$ (i.e., $Frac'_i - Frac_i$) and rank them from lowest to highest. $order(i)$ represents the rank of application $i$, starting from 0 to $N - 1$. Requests for lower CPU fractions than previous epoch are granted first. After that, the current extra CPU share is obtained, which is the sum of the fraction of CPU idle time and the released CPU fractions. The requests for higher CPU fractions are then considered in order. If the request is less than the current average of extra CPU share (i.e., $Frac'_i - Frac_i \leq Frac_{extra}/(N - order(i))$), the request is satisfied and $Frac_{extra}$ is updated. Up to this point, the reassignment is simple as $Frac''_i = Frac'_i$. 

Figure 4.5. CPU fractions when executing a workload of two I/O-intensive applications.
Following these steps, all the applications yet to handle require more CPU fractions than the equal share would provide. One way to handle the rest allocations is to continue favoring smallest requests and ignore the large requests. However, it hurts the fairness especially if the reason for large fraction request is because the reference target has been missed by a large margin. The approach we take is based on how effectively each application could use the extra CPU fraction. To quantify this property, we first define relative error as follows:

\[ e_i = \frac{IPC_i - IPC_{ref,i}}{IPC_{ref,i}}. \] (4.5)

We normalize the error so that it can be compared across applications. With the relative error, we define the fraction sensitivity for application \( i \) as:

\[ sensi_i = \frac{\Delta e_i}{\Delta Frac_i}. \] (4.6)

Here, \( \Delta e_i \) is the change in relative error and \( \Delta Frac_i \) is the change in measured CPU fraction. The fraction sensitivity is then used as the weight factor to allocate the extra CPU fraction as follows:

\[ Frac''_i = Frac_i + Frac_{extra} \ast \frac{sensi_i}{\sum_j sensi_j}. \] (4.7)

In summary, the fraction reassignment procedure grants requests for increase in CPU fractions when the request is comparable to the equal share of extra resource. Meanwhile, demanding requests are isolated and handled in a fair way based on how effectively extra resources can be utilized.

**• Deriving Time-slices and Nice Values.** Once the final assignments of new fractions (\( Frac'' \)) are performed, we can choose the new quantum accordingly. One of the benefits of applying control theory is to provide stability and to avoid too much oscillation. In our case, this means avoiding dramatic changes in the base time-slices and nice values. To achieve this goal, we propose a moving-average filter and apply it when deciding the new time-slices and associated nice values for applications with QoS specified.

The rationale behind the moving-average filter is to consider the history values for the output variables (time-slice values in our case). We keep track of the base
time-slice values that have been chosen in the past by maintaining a variable $m_i$ for application $i$, which is defined as:

$$m_{i,t} = \beta m_{i,t-1} + (1 - \beta)q_{i,t-1}.$$  \hfill (4.8)

$m_{i,0}$ is initialized as $q_{i,0}$. $\beta$ is a parameter that can be adjusted to reflect the importance of past values in different length of history. If $\beta = 1$, $m_i$ always equals to $m_0$ and only the initial nice value is reflected. If $\beta = 0$, $m$ is just the last chosen nice value. In our study, we use 0.8 as the default $\beta$ value.

For application $i$, we can measure its CPU fraction during the past interval ($Frac_{i,t-1}$) with a time-slice of $q_{i,t-1}$ before the execution of the $t$-th interval. Based on the reassigned desired fraction ($Frac''_{i,t}$), we can predict the time-slice for the $t$-th interval ($\phi_{i,t}$) using the models introduced earlier.

With these variables set up, the new base time-slice is calculated as follows:

$$q_{i,t} = q_{i,t-1} + \phi_{i,t} - m_{i,t}.$$  \hfill (4.9)

Here, $q_{i,t-1}$ is the last base time-slice value for application $i$; $\phi_{i,t}$ is the predicted time-slice value; and $m_{i,t}$ is a variable that takes history into account. With the new time-slice, the corresponding priority (or nice value) can be selected for this application. Note that the base CPU time-slice can only be one of the discrete values that are associated with priorities.

### 4.3 Experimental Setup and Evaluation

We used the Simics toolset [12] to implement our approach and perform our experiments. Table 4.1 gives the major simulation parameters used in this study with their default values. Most of our experiments use one-core processor and determine the time-slices for the serial applications that share the core. In section 4.3.3, we also present results when a four-core machine is used. The operating system running on the simulated machines is Fedora Linux 2.6.15.

The easiest way to control the base CPU time-slice used in scheduling is to use the `renice` system command with the process id. However, running such system command incurs extra performance overhead and delay in the controller, especially
when we need to modify the nice values for multiple processes. For example, we observe that more than 30% of CPU time could be spent in calling `renice` multiple times within an epoch of 10M instructions. Therefore, we modified the scheduler in Linux kernel to add a specialized function so that nice values can be set at once with much less overheads.

Table 4.1. Major simulation parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>1 or 4</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>2GHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32K, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32K, 2-way</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>4MB, 16-way</td>
</tr>
<tr>
<td>L1 Access Latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L2 Access Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory Access Latency</td>
<td>260 cycles</td>
</tr>
</tbody>
</table>

The benchmarks we used in this study are selected from SPEC CPU 2006 [104] and PARSEC 2.1 [70]. For all the experiments presented below, we fast forwarded the first 4 billion instructions and then warmed up caches for the next 1 billion instructions. As our QoS targets are specified as IPC, we classify the SPEC 2006 benchmarks into several categories according their IPC characteristics. We first run each benchmark on its own and measure the IPC value and the number of cycles
the benchmark is in active status. If the IPC is relatively low and the number of active cycles only takes a small fraction of the total cycles, this usually means that application has a lot of I/O operations or other events blocking execution. If most CPU cycles are spent executing the benchmark, the benchmark is often CPU-intensive. Based on this criteria, our applications can be divided into “I/O intensive” and “CPU intensive” classes. After that, we studied the IPC variations for each benchmark by running it for 100 epoches with 10M instructions in each epoch. Based on the calculated variance values, the benchmarks can then be divided into “high variation” and “low variation”. The benchmarks listed in Table 4.2 are the representative ones we select from SPEC 2006 to evaluate our approach. Note that the classification is relative and may be different if we select different portion of an benchmark. For the multi-threaded PARSEC benchmarks, we used the OpenMP implementations of blackscholes and bodytrack for our experiments on four-core machines. Different threads of the same OpenMP application have similar behaviors when running on their own.

<table>
<thead>
<tr>
<th>CPU intensive</th>
<th>High Variation</th>
<th>Low Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O intensive</td>
<td>cactus</td>
<td>milc</td>
</tr>
</tbody>
</table>

Table 4.2. Selected SPEC2006 benchmarks.

Figure 4.8. IPC values for bzip2 and milc in the default scenario.

Figure 4.9. IPC values after applying the controller for bzip2 and milc.

Figure 4.10. IPC values for bzip2, gcc and libq in the default scenario.

Figure 4.11. IPC values for bzip2, gcc and libq after applying our controllers.

Figure 4.12. IPC values for a four-application workload in the default scenario.

Figure 4.13. IPC values for a four-application workload after applying our controllers.
4.3.1 Tracking the QoS Targets

We have shown the different execution scenarios for a two-application workload involving bzip2 and gcc in Figure 4.1. We start the evaluation of our approach using the same workload with several experiments to control the IPC values. Figure 4.6 provides the IPC variations, actual CPU fractions and nice values assigned when we assign 5% and 10% as the reference IPC for bzip2 and gcc, respectively, to illustrate the function of service differentiation. Recall from Figure 4.1(a) that bzip2 has higher IPC than gcc in the default case. As we can see in Figure 4.6(a), the IPC values for both applications are above the reference values most of the time. The actual CPU fractions in Figure 4.6(b) vary dynamically to affect the IPC values. For example, the CPU fraction of gcc is larger than that of bzip2 because gcc needs to be favored in order to track the reference IPC. Similarly, when the IPC value of bzip2 falls below the reference value, its CPU fraction gets increased so that the QoS requirement could be met. These CPU fractions are controlled by applying different nice values shown in Figure 4.6(c).

We now evaluate our control framework when the reference targets are set low and thus excess resources exist. The reference IPC values are set at 8% for both bzip2 and gcc. Figure 4.7 depicts the results for IPC values, CPU fractions and nice values, respectively. The IPC values for both applications are close to 10%, which is above the reference input. This shows that the excess resource are utilized properly when the reference targets are low. Unlike the results in Figure 4.6(c), the nice values for both applications change dynamically. Specifically, although the IPC value of bzip2 falls below reference input, the controller tries to increase the CPU fraction by decreasing its nice value. This also illustrates that our approach is of “best-effort” type.

As both bzip2 and gcc are CPU-intensive benchmarks, we replaces gcc in the two-application workload with milc which has been shown to be very I/O-intensive. Figure 4.8 plots the IPC values for bzip2 and milc in the default scenario. Compared to Figure 4.1(a), one can tell that the IPC of bzip2 becomes much less when running with the I/O-intensive benchmark. We set the IPC targets as 10% for bzip2 and 1% for milc. Figure 4.9 illustrates the IPC results after applying our controllers. We see that the IPC target for bzip2 is well tracked whereas milc keeps similar IPC values. This is because the CPU fraction for bzip2 is successfully ad-
justed at runtime but the target for milc is unsatisfiable just through varying nice values.

Figure 4.10 shows the IPC values for a workload with three benchmarks (bzip2, gcc and libq) when executing together without tuning time-slice. Note that these three benchmarks all have “high variations” in IPC values based on our benchmark characterization and as exhibited in Figure 4.10. Figure 4.11 illustrates the IPC values (for the same workload) using our approach when we target an IPC value of 3% for all three applications. We see that the IPC targets are satisfied except at a few data points and the IPC fluctuation range is also decreased.

Figures 4.12 and 4.13 present the IPC values for a four-application workload in the default case and when applying our controllers. The four applications used in this experiment are calcu, bzip2, omnet, and sjeng. The IPC references are 7% for calcu and bzip2 and 5% for omnet and sjeng, respectively. Compared to Figure 4.12, the IPC curves in Figures 4.13 are separated into two groups with each group centers around corresponding reference input. One can also notice the successful service differentiation by comparing the IPC values of bzip2 and omnet across these two figures.

4.3.2 Selection of QoS Targets

The selection of QoS targets is an important issue in managing the quality of service in almost all systems with shared resources. In this study, we use a single IPC value for each application. To choose a reasonable IPC value, one can run the application by its own first and obtain some average value out of the execution profile. Another alternative would be to change the QoS target at runtime as well. This can be done by sampling the IPC value for the application during its course of execution. However, the measured value could be dependent on the runtime characteristics of co-runners. If the sampling frequency is low, one may let the application run on its own to get the sampling data without introducing too much overheads.
4.3.3 Extending to Multicore Architectures

So far our study has focused on single-core processor. Our dynamic time-slice tuning framework can be extended to multicore architectures as well. In the multicore case, the desired IPC values can still be used as QoS targets and the CPU fractions on each core need to be managed among competing threads. There are two cases when considering multicore processors.

The first scenario is to run multiple single-threaded applications on multiple cores. In this case, additional work comparing with single-core scenario is to provide an ideal mapping from applications to cores. For example, we would like to distribute applications with large CPU fraction requirements across cores instead of mapping them to one place. If one core is over-subscribed and the other is under-subscribed, we can migrate threads so that the requirements are more balanced and thus easier to satisfy. This sounds similar to the load balancing issue that has been widely studied in the literature. However, the load is treated differently in our case because the requests for CPU fractions are affected by the QoS targets instead of execution characteristics. Note that such mapping can be dynamic at runtime. Once a specific mapping is selected, each core can use the same control framework we proposed in Figure 4.2 for those processes that are mapped onto it.

The second scenario is to have multi-threaded applications running on the multicore. It becomes more challenging when threads of the same application are mapped to different cores. These threads usually have synchronizations during execution but run at different speed. As a result, performance of an multi-threaded application often depends on some threads which runs slower that others. For example, Bhattacharjee and Martonosi [105] studied the criticality of different threads of multi-threaded applications and proposed an predication scheme based on memory hierarchy statistics. Muralidhara et al. [106] also observed the performance difference among multiple threads of the same application and built intra-application cache partitioning.

To manage the QoS for multi-threaded application, the overall target needs to be translated into targets for individual threads. Figure 4.14 illustrates this process. We assume $M$ multi-threaded applications are running on $N$ cores and each application has one thread on each core. $IPC_{ref,i}$ is the reference IPC for multi-threaded application $i$. This performance target is passed to application
controller which defines the target IPC for individual threads on different cores. $IPC_{\text{ref},i,j}$ means IPC target for application $i$’s thread on core $j$. At the core level, each core can use the framework shown in Figure 4.2. The actual IPC values for all the threads of an application are fed back to the application controller. $IPC_{i,j}$ stands for the IPC values measured at runtime for application $i$’s thread on core $j$. In the general case, applications with different thread-level parallelism (from single-threaded to N-threaded) are executed on the multicore. The illustrated framework still works with IPC target set as 0 for non-existent threads.

Application controller needs to compare the defined target and measured IPC value for individual threads and updates the target IPC values at runtime. First, the changes in measured IPC values and control errors (i.e., difference between the target and measured IPC values) indicate the sensitivity of each thread under our control method. Second, the IPC values of other threads within the same application can be reflected in the new target IPC. The design of the application controller has many options. One straight-forward way is to focus on the slowest thread in terms of IPC. This is based on the observation that fast threads need to wait for the slow threads at synchronization points. In this case, application controller tried to maintain the IPC values of different threads around the same value, which might be lower than the target. An improvement to such application controller could provide hints to the core level controller with an IPC target range. If other applications could utilize the extra CPU fraction, the lower limit of the IPC target can be selected. Otherwise, the IPC target can be set up to the upper limit. A study of different application controller is not in the scope of this chapter.

Figure 4.14. Feedback control framework for multiple multi-threaded applications on multicore.
We experiment with two parallel applications (blackscholes and bodytrack from PARSEC 2.1) concurrently executing on a simulated four-core machine. Each application runs four threads that are pinned onto four cores, i.e., each core runs a thread from each application. Figure 4.15 presents the IPC values for the thread of bodytrack on core 0 both in default scenario and when targeting at an IPC of 0.1 with our controllers. We see that the IPC target for the bodytrack thread is tracked properly. Threads on other cores showed similar behavior.

![Figure 4.15. IPC values of the bodytrack thread on core 0 with IPC target at 0.1.](image)

### 4.4 Related Work

We have compared recent studies that proposed various QoS schemes and explained the difference between these studies and our work in the beginning of this chapter. In this section, we briefly discuss the related work on managing CPU time-slice, contention-aware scheduling, and applying control theory in computing systems.

Fedorova et al. [68] described a software solution to improve performance isolation in multicores by changing the thread time-slice at the OS level based on expected IPC derived from the hardware performance counters. Giné et al. [107] introduced an algorithm to adjust the time-slice dynamically so that the cache miss rate is decreased in time-sharing distributed systems. Thompson et al. [108] analyzed the implementation of round-robin and inferred a proper time-slice considering the context-switch overhead.

Zhuravlev et al. [98] investigated scheduling techniques to mitigate performance degradation due to contention in shared resource. Their focus is more on the grouping/mapping of threads and assume one thread per core. In comparison, we study the competition of CPU time inside a processor core. Snively and Tullsen [53] demonstrated that the performance of a multi-threaded processor is
sensitive to the set of co-scheduled processes and used sampling of the processes to capture the thread interactions. Algorithms for fair scheduling in general-purpose OS have also been discussed in the literature. Chandra et al. [109] presented a weight re-adjustment scheme to reduce unfairness. Li et al. [110] proposed distributed weighted round-robin algorithm that achieves accurate fairness and high performance.

Control theory has been successfully applied to a wide range of problems in the computer systems [100]. Steere et al. [111] proposed a feedback-based controller that predicts application progress and assigns CPU proportion and period to threads. They focused on special applications that have throughput requirement but no deadlines. Compared to their work, our work provides different levels of service quality and differentiation. Other applications of control theory to computing systems include realtime system scheduling [112], power and temperature management [113] and cache management [95].

4.5 Conclusions

In this chapter, we investigate how to control the quality of service (QoS) for time-sharing workloads by dynamically adjusting the base CPU time-slice. We build a formal control feedback loop to help adjusting the CPU time-slice (through changing process priorities). Our experimental results show that the proposed framework can successfully track the quality of service targets and provide service differentiation.
Chapter 5

Future Work and Conclusions

This last chapter provides discussion on potential future work that can be extended from the studies presented in this dissertation. A summary of contributions is also given in the end.

5.1 Future Work

Resource management on chip multi-processor is a broad research topic that many researchers have worked on in recent years. Although there is a large amount of literature which helps our understanding in this area, there are still a lot of interesting questions need further study. This dissertation focuses on runtime adaptations in the resource management and provides insights into how to improve energy efficiency and quality of services. Some potential future research studies that can be built on top of this work are discussed in the following.

5.1.1 Heterogeneous Systems

All of the studies in previous chapters have assumed homogenous architecture systems. In other words, the different processor cores in a CMP have same specifications. While this assumption fits most CMPs in the current marketplace, heterogeneous systems have drawn a lot of interests as well and emerge in quite a few system designs such as such as IBM Cell [10]. An another example is to include GPU into general computing. In this case, programs or kernels can run
in either GPU or CPU. How to manage these heterogeneous systems brings a new dimension of challenges.

On the other hand, heterogeneity could also exist in homogeneously designed systems due to process variations. Process variation has been identified as one of the key design challenges in the future semiconductor industry. As the process technology goes into the deep sub-micron regime, it is becoming increasingly difficult to control critical transistor parameters such as gate-oxide thickness, channel length, and dopant concentration. As a result, these parameters may have different values than nominal, which may, in turn, lead to both power and timing variability across identically-designed components.

Adapting the application execution to the system heterogeneity is an interest area to explore. As an example, I studied core-to-core variations caused by the systematic with-die process variation, where different cores have different performance and power characteristics. We target at a 16-core (4X4) chip multiprocessor (CMP) architecture with four rows and four cores in each row as illustrated in Figure 5.1. We assume that cores in different rows have different power and performance characteristics whereas cores in the same row are similar. Specifically, those cores in the top rows are faster but leakier, and the cores in the bottom rows are slower but less leaky. We explored the thread mapping schemes to optimize performance, energy consumption and energy-delay product. Figure 5.2 provides a summary of
different mapping methods with four applications each running on four cores. Applications are chosen from NAS NPB 3.2 benchmark suite and SPEC OMP 2001 benchmark suite. In each group of bars, the first bar shows the EDP using a process variation aware scheduling scheme that employs integer linear programming (ILP). Significant EDP savings can be obtained by applying the process variation aware scheduling scheme.

One of the traditional ways to utilize heterogeneous system is to hide the difference in hardware and present a uniform interface to the software. Our results show that thread mapping in CMPs should be aware of the core-to-core variation and can optimize both performance and energy consumption globally. Once the hardware heterogeneity is exposed to software, runtime systems can take advantage of the flexibility to optimize the execution.

### 5.1.2 Beyond Single Level Resource Management

Many resources are shared on a chip multi-processor, including processor cores, shared L2/L3 caches, shared memory bandwidth, shared memory and I/O interfaces. Most of the studies in this dissertation consider one level of resource. For example, in multi-application workloads, we focused only on partitioning cores or CPU time-slices in this work. Considering multiple levels of resources together could improve the performance further. In these cases, the L2 banks in the CMP can be partitioned across applications as well[64, 65, 66, 96, 103]). The resource requirements for different applications vary. Therefore, adjusting the allocations at certain level may not be effective for all the applications. When multiple resources are considered together, we not only have more alternatives to better serve one application, but also can improve the utility of the resources.

On the other hand, even when managing one type of resources, there are different ways to control the resources. As an example, our discussion in Chapter 3 partitions processor cores without considering voltage/frequency scaling. Allowing voltage/frequency scaling can help to further optimize the W-EDPG metric. Given the execution latency and energy consumption of a thread under maximum core frequency, it is possible to predict its execution latency and energy consumption under lower frequency using an analytical model such as that adopted by Isci et al.
(which assumes that the ratio between performance degradation and power savings is 1 : 3). Based on these information, different core partitioning may be selected. Similarly, for the time-slice tuning, instead of selecting different time-slices for the threads running simultaneously. Another way to manage the shared CPU cycles is to change the frequency that each thread is allowed to run for its time-slice. Considering multiple ways when managing one type of resources could improve the utility further.

When multiple resources are shared, it would be very helpful to identify which resource is more critical and which one is more sensitive. The benefits brought by adjusting the allocation for each level resource have limits. For instance, allocating more cache space can improve the IPC in most cases, but if more enhancements are required, we may consider giving it more cores, longer time-slices or increase the core frequencies among other adjustments.

5.1.3 Other Objectives and Metrics

Apart from the energy efficiency and quality of services, there are other important objectives that an adaptive resource management scheme can focus on. Performance is always one of the most important optimization target. Our results have shown possible speedup improvement along with the EDP reductions. More accurate performance predication model such as [105] can help making optimal decisions in the resource management. Other optimization objectives such as reliability also draw a lot of interests. For example, Coskun et al. [115] proposed a reliability-aware job scheduling and power management framework to affect the processor lifetime. Managing the usage of different components in one resource can avoid stressing one component too much and causing it to fail. When extra resource is available, certain hardware redundancy can also be implemented to provide system reliability. There are a lot of tradeoffs to study in order to make these schemes beneficial.

The metrics used in this dissertation include application-level EDP for energy efficiency and IPC for the quality of services. Other metrics might be more appropriate in certain cases and are worthy studying. For example, when considering the EDP for multiple-application workload in Chapter 3, the average of EDP values for
individual applications is used. Another alternative is to multiply the total energy consumption to the execution time for the whole workload to finish. As another example, for some scientific applications, floating-point operating per cycle or per second may make more sense than IPC. Optimizing one metric under constraints in other metrics is also important, such as minimizing the energy consumption under performance bounds. The studies presented in this dissertation discussed about these issues and a further study may make the adaptive approach more appealing in other areas.

5.2 Summary of Contributions

The main contribution of this dissertation is the discussion and evaluation of an adaptive approach in resource management to improve the energy efficiency and quality of service in a chip multi-processor based environment.

The helper thread concept is adopted to track the application executions in our resource management framework. Based on the program structures, we identify the appropriate granularity for adaptations. Curve fitting models are used to infer ideal resource requirements such as the number of cores, number of threads, and voltage/frequency levels. These predictions are based on the characterization collected at runtime. For multi-application scenarios, our approach partitions the resource effectively and improves the overall performance.

A software approach for quality of services is proposed by dynamic tuning the time-slices of multi-application workloads in time-sharing operating systems. In order to satisfy the QoS requirements specified by users, a formal feedback control framework is built to dynamically tune the process nice values and associated CPU time-slices for simultaneously running applications. Experimental results show that the proposed framework can successfully track the quality of service targets and provide service differentiation.
Bibliography


[63] Chen, S. et al. (2007) “Scheduling threads for constructive cache sharing on CMPs,” in Proc. of SPAA.


[66] Lin, J. et al. (2008) “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems,” in Proc. of HPCA.


Vita
Yang Ding

Yang Ding joined the PhD program in the department of Computer Science and Engineering at Penn State University in August 2005. His research interests include performance and energy optimizations using techniques in both computer architecture and compiler levels. He received his B.S. (2002) in computer science from Peking University, China and M.S. (2003) in computer science from National University of Singapore.