Usage-pattern-based scheduling schemes for power optimization in multi-core systems

Hou, Zhaoqi

2017

Hou, Z. (2017). Usage-pattern-based scheduling schemes for power optimization in multi-core systems. Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/72031>

<https://doi.org/10.32657/10356/72031>

**Usage-Pattern-Based Scheduling Schemes for
Power Optimization in Multi-core Systems**

HOU ZHAOQI

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2016

Acknowledgements

I would like to express my deep gratitude to my supervisor, Prof Jong Ching Chuen, for his patient guidance, kind mentoring and time devoted to help me with the revision of the thesis and the research work in the past years.

My sincere thanks to my co-supervisor Prof Andreas Herkersdorf, whose exclusive knowledge of embedded and many-core systems had helped me overcome many obstacles in the progress of the research, the cold winter, and hot summer without air conditioning system.

At last, to my beloved wife, for her tenderness and acquaintance that brings me the tremendous serendipity.

TABLE OF CONTENTS

ABSTRACT	IV
TABLE OF FIGURES	V
TABLE OF TABLES	VIII
TABLE OF ACRONYMS.....	IX
CHAPTER 1 INTRODUCTION	- 1 -
1.1 MOTIVATION AND OBJECTIVES	- 1 -
1.2 MAIN CONTRIBUTIONS	- 3 -
1.3 ORGANIZATION OF THE THESIS	- 6 -
CHAPTER 2 LITERATURE REVIEW.....	- 7 -
2.1 PARALLELISM AND MULTI-CORE PROCESSORS	- 7 -
2.2 POWER MANAGEMENT APPROACHES	- 9 -
2.3 SYSTEM PERFORMANCE METRIC QUANTIZATION	- 13 -
2.4 CPU TEMPERATURE AND POWER DISSIPATION.....	- 16 -
2.5 POWER MANAGEMENT IN HETEROGENEOUS ARCHITECTURES	- 18 -
2.6 NEURAL NETWORK APPLICATIONS AND OPTIMIZATION.....	- 20 -
CHAPTER 3 USER PATTERN AWARE SCHEDULING SCHEMES	- 24 -
3.1 INTRODUCTION	- 24 -
3.2 PROPOSED POWER MANAGEMENT APPROACH.....	- 25 -
3.2.1 <i>User Power Profile Recognition</i>	- 26 -
3.2.2 <i>Thread Interference Power and Thread Solitary Power Model</i>	- 27 -
3.2.3 <i>Monitor the Scheduling Activities</i>	- 29 -
3.2.4 <i>Updating AppSet Table</i>	- 30 -
3.2.5 <i>Quiet Intervals</i>	- 32 -
3.2.6 <i>An Illustrative Scenario</i>	- 33 -
3.2.7 <i>User Power Profile Recognition and Computation Space</i>	- 36 -
3.3 APPLICATION CATEGORIZATION AND SCHEME ALLOCATION	- 38 -
3.3.1 <i>Power Management Schemes</i>	- 39 -
3.3.2 <i>Thread Interference Power and Thread Solitary Power Optimization</i>	- 40 -
3.3.3 <i>Scenario Decision</i>	- 44 -
3.4 SYSTEM SIMULATION AND RESULT VERIFICATION	- 47 -
3.4.1 <i>System Model</i>	- 47 -
3.4.2 <i>Virtual Users and Simulation Scenarios</i>	- 49 -
3.4.3 <i>Simulation Procedures</i>	- 49 -
3.4.4 <i>Results and Discussion</i>	- 51 -
3.5 CONCLUSION AND FURTHER WORKS	- 54 -
CHAPTER 4 APP USAGE PATTERN AWARE LINUX SCHEDULER.....	- 56 -
4.1 INTRODUCTION	- 56 -
4.2 PMP QUANTIZATION, PDP MODELING AND THERMAL EQUATION	- 58 -
4.2.1 <i>Performance Metric Quantization</i>	- 59 -
4.2.2 <i>Thermal Hotspot and Power Dissipation Quantization</i>	- 69 -

4.2.3 Power Dissipation Problem Categorization.....	- 74 -
4.3 STRUCTURE OF THE RE SCHEDULER	- 79 -
4.3.1 Re Scheduler Function Blocks	- 79 -
4.3.2 Re Scheduler Scheduling Schemes	- 89 -
4.4 MULTI CLUSTER SETUP.....	- 93 -
4.4.1 Three-Clustered CPU Setup.....	- 93 -
4.4.3 Simulation Tool Chain.....	- 97 -
4.5 RESULTS	- 98 -
4.6 CONCLUSION	- 108 -
CHAPTER 5 DEEP FUNCTION NETWORK AIDED SCHEDULER	- 110 -
5.1 OVERVIEW OF SCHEDULER IN DNN STRUCTURE.....	- 110 -
5.1.1 ANN and DNN in Mobile Computing Environment	- 110 -
5.1.2 Reduction of Computation Overhead Generated in Re Scheduler.....	- 110 -
5.1.3 Online and Offline Learning of the Re DFN.....	- 112 -
5.1.4 Chapter Organization.....	- 113 -
5.2 NEURAL NETWORK FUNCTION BLOCKS AND LEARNING ALGORITHMS	- 113 -
5.2.1 RBM and DBN in Re DFN.....	- 114 -
5.2.2 Autoencoder and Stacked Autoencoder in Re DFN	- 117 -
5.2.2 Recurrent Neural Network in Re DFN	- 120 -
5.3 LAYERS AND COMPONENTS OF THE RE DFN.....	- 121 -
5.3.1 Relations of the Re DFN and the Original Re Scheduler	- 122 -
5.3.2 Layer-wise structure of Re DFN	- 124 -
5.4 RE DFN OPTIMIZATION	- 136 -
5.4.1 Sensitivity Analysis and Neural Network Pruning	- 136 -
5.4.2 Quantization and Weight Sharing.....	- 139 -
5.4.3 Sparse Matrix Storage Format.....	- 141 -
5.5 RESULTS	- 143 -
5.5.1 Training Error and Test Error.....	- 143 -
5.5.2 Size Reduction and Accuracy after Optimization.....	- 150 -
5.5.3 Power Reduction	- 152 -
5.6 CONCLUSION	- 155 -
CHAPTER 6 CONCLUSIONS AND RECOMMENDATIONS.....	- 157 -
6.1 TIP-TSP MODEL.....	- 158 -
6.2 PMP QUANTIZATION AND PDP DETERMINATION	- 158 -
6.3 NEURAL NETWORK TIME SERIES DATA CLASSIFICATION AND OPTIMIZATION	- 160 -
6.4 FUTURE WORKS.....	- 161 -
AUTHOR'S PUBLICATIONS.....	- 163 -
REFERENCE.....	- 164 -

Abstract

This thesis discusses the scheduling schemes with app usage pattern awareness for power dissipation optimization in mobile multi-core computing systems. The power dissipation of mobile multi-core computing systems becomes crucial as the power wall symptom arises. One way to resolve the power wall issue is to manage the system resource with power awareness focus. Three main areas of work are presented in the thesis to address the power dissipation issues.

First, three scheduling schemes are proposed to decompose the instantaneous power dissipation into different components to model the redundancies in application inflicted power dissipation. The decomposition identifies the manageable power components as the Thread Interference Power which exists among the context switching of multithreading applications. Different amounts of Thread Interference Power dissipate between different combinations of adjacent threads. In this thesis the application combination and power dissipation relationship is studied, identified at system runtime. Application thread scheduling schemes are designed based on the assumption that the usage pattern of a mobile system user is periodical, hence the future application sequence and possible power dissipation are predictable based on the past performance recorded. The scheduling schemes presenting in the thesis make scheduling decisions based on the instantaneous system application combinations and the possible incoming application predicted. The scheduler reorders the application threads according to the amount of Thread Interference Power dissipated. The scheduler is testified in a simulated environment built with GEM5 and McPAT. The simulation results show that the adoption of usage pattern aware scheduling method is capable of reducing the total system power dissipation up to 19% compared to the total system power dissipated without applying the proposed method.

To incorporate the proposed scheduling schemes in a Linux scheduler, the quantization method of the Thread Interference Power dissipation with system monitored performance events is studied. Multiple power dissipation related problems are studied to generalize the function of the scheduling schemes as functions of a scheduler. Mutual cache evictions or invalidations by parallel processes, corresponding data movements, context switching overheads and code serialization bottlenecks due to unfortunate scheduled synchronization / locking mechanisms on shared access resources are major contributors to the excessive computing power dissipation of modern chip multiprocessor (CMP) architectures. The proposed Linux scheduler manages above mentioned problems based on user-specific application usage pattern identified in the Thread Interference Power model. To extend the scheduling capacity of the proposed scheduler, novel Dynamic Voltage and Frequency Scaling (DVFS) techniques are developed for ARM asymmetrical System-on-Chip (SoC) designs and tested in a three clustered CPU setup. The result of the proposed scheduler shows a maximum of 22.2% improvement in the power dissipation reduction over the native Global Task Scheduler (GTS).

Last but not least, to better the prediction based power dissipation monitoring mechanism in the proposed scheduler, neural network algorithms are adopted to solve the problem with non-parametric assumption. Each power component is sampled individually over a period of time and undergoes analysis as a time series. Thread Interference Power components are then identified and isolated from the time series power dissipation with the aid of a system of neural network algorithms including Autoencoder, Restricted Boltzmann Machine and recurrent neural network. With the aid of the neural network implementation the proposed scheduler reduced the computation overhead by 73.3% and power dissipated by a maximum of 8.1%.

TABLE OF FIGURES

FIGURE 3.1 OVERVIEW OF THE MANAGEMENT SCHEMES	- 26 -
FIGURE 3.2 MULTITHREADED APPLICATION THREADS	- 26 -
FIGURE 3.3 TIME SLICE SEQUENCE 1 AND 2	- 28 -
FIGURE 3.4 AN ILLUSTRATIVE SCENARIO WITH QUIET INTERVALS	- 33 -
FIGURE 3.5 <i>AppSet Table</i> AT T1	- 34 -
FIGURE 3.6 <i>AppSet Table</i> AT T2	- 34 -
FIGURE 3.7 <i>AppSet Table</i> AT T3	- 35 -
FIGURE 3.8 <i>AppSet Table</i> AT T4	- 35 -
FIGURE 3.9 <i>AppSet Table</i> AT T5	- 35 -
FIGURE 3.10 <i>AppSet Table</i> AT T6	- 36 -
FIGURE 3.11 EQUALLY DISTRIBUTED TIME SLICES AT MULTITHREADING INTERVALS	- 38 -
FIGURE 3.12 COMPARISON BETWEEN THE NATIVE SCHEME AND PROPOSED HOLD SCHEME	- 39 -
FIGURE 3.13 AN EXAMPLE OF THE DEDICATED SCHEME	- 40 -
FIGURE 3.14 AN EXAMPLE OF THE TOTAL PARALLELISM SCHEME	- 40 -
FIGURE 3.15 THE SCHEDULER QUEUE AT T _M	- 42 -
FIGURE 3.16 THE TIP AND TSP IN DIFFERENT <i>AppSets</i> SCHEDULING SEQUENCES	- 43 -
FIGURE 3.17 THE NATIVE AND HOLD SCHEME	- 45 -
FIGURE 3.18 THE REALIZATION OF PARALLELISM SCHEMES	- 45 -
FIGURE 3.19 AN EXAMPLE OF THE PREDICTION COMPUTATION	- 46 -
FIGURE 3.20 THE ESTABLISHED GEM5 SETUP	- 48 -
FIGURE 3.21 THE GEM5 AND MCPAT INTERFACE SETUP	- 48 -
FIGURE 3.22 THE BUSINESS USER NATIVE SCHEME SCHEDULING SCENARIO	- 50 -
FIGURE 3.23 THE BUSINESS USER ALL SCHEMES SCHEDULING SCENARIO	- 50 -
FIGURE 3.24 TIP AND TSP OF GENERAL USER	- 51 -
FIGURE 3.25 TIP AND TSP OF MEDIA USER	- 52 -
FIGURE 3.26 TOTAL POWER DISSIPATED OVER TIME OF BUSINESS USER	- 52 -
FIGURE 3.27 TIP-TSP DISSIPATION BY APP OVER TIME OF GENERAL USER	- 53 -
FIGURE 3.28 TIP-TSP DISSIPATION BY APP OVER TIME OF MEDIA USER	- 53 -
FIGURE 3.29 TIP-TSP DISSIPATION BY APP OVER TIME OF BUSINESS USER	- 54 -
FIGURE 3.30 PROPOSED SCHEDULING SCHEMES WITH DVFS INCORPORATED	- 55 -
FIGURE 4.1 PMP NORMALIZATION AND PDP BASED TIP-TSP DECOMPOSITION	- 59 -
FIGURE 4.2 MEMORY UNIT ACCESS LATENCIES	- 61 -
FIGURE 4.3 CACHE HIT TIME OF A SEQUENTIAL WORKING SET	- 61 -
FIGURE 4.4 CACHE MISS RATE OF A RANDOM WORKING SET	- 62 -
FIGURE 4.5 NUMBER OF TLB MISS AND PAGE WALK CYCLES	- 64 -
FIGURE 4.6 TLB MISS RATE OF DIFFERENT WSS	- 64 -
FIGURE 4.7 BRANCH RESOLUTION IN DIVERGENCE	- 65 -

FIGURE 4.8 BRANCH RESOLUTION IN CONVERGENCE.....	- 65 -
FIGURE 4.9 BRANCH PREDICTION MISS RATE OF DIFFERENT BENCHMARKS	- 66 -
FIGURE 4.10 PROCESS OF A CONTEXT SWITCH.....	- 66 -
FIGURE 4.11 CONTEXT SWITCHING COST WITHIN THE CPU CORE	- 67 -
FIGURE 4.12 CONTEXT SWITCHING COST ACROSS CPU CORES.....	- 68 -
FIGURE 4.13 CORTEX A57 CPU CORE POWER DISSIPATION AT DIFFERENT OPPS AND TEMPERATURE	- 70 -
FIGURE 4.14 CPU CORE TEMPERATURE CHANGE SWITCHING FROM 1.2 GHZ TO DIFFERENT OPP	- 72 -
FIGURE 4.15 TEMPERATURE RISE FROM 1.2-2.1 GHZ	- 73 -
FIGURE 4.16 POWER DISSIPATION RISE FROM 1.2-2.1 GHZ	- 73 -
FIGURE 4.17 PDP THRESHOLD T OF DIFFERENT POWER DISSIPATION PROBLEM.....	- 76 -
FIGURE 4.18 RE SCHEDULER SCHEDULING STEPS AND FUNCTION BLOCKS	- 79 -
FIGURE 4.19 PER CORE <i>AppSet Table</i>	- 80 -
FIGURE 4.20 <i>AppSet</i> AT THE END OF THE FIRST EPOCH.....	- 82 -
FIGURE 4.21 <i>AppSet</i> AT THE END OF THE SECOND EPOCH.....	- 83 -
FIGURE 4.22 <i>AppSet</i> AT THE END OF THE THIRD EPOCH.....	- 83 -
FIGURE 4.23 <i>AppSet</i> AT THE END OF THE FOURTH EPOCH.....	- 84 -
FIGURE 4.24 PER CORE COST TABLE	- 85 -
FIGURE 4.25 APPSET HIERARCHY OF A MULTI PATH <i>AppSet Table</i>	- 86 -
FIGURE 4.26 FORWARD <i>AppSet Table</i> WALK	- 86 -
FIGURE 4.27 BACKWARD <i>AppSet Table</i> WALK	- 86 -
FIGURE 4.28 <i>AppSet</i> HIERARCHY IN CORE 1	- 88 -
FIGURE 4.29 <i>AppSet</i> HIERARCHY IN CORE 2	- 88 -
FIGURE 4.30 RUN QUEUE CONTENT AND PREDICTION OF NEXT INCOMING APPLICATION THREAD	- 88 -
FIGURE 4.31 DECOMPOSED <i>AppSeti</i> AFTER FORWARD WALKING	- 89 -
FIGURE 4.32 DIFFERENT PATH THAT LEADS TO RECONSTRUCT <i>AppSeti</i> AT BACKWARD WALKING.....	- 90 -
FIGURE 4.33 <i>AppSetd</i> IN RUN QUEUE.....	- 91 -
FIGURE 4.34 <i>AppSetd</i> WITH NORMAL TIME SLICE AND WITH SUPER TIME SLICE IN EACH EPOCH	- 91 -
FIGURE 4.35 DEDICATED SCHEDULING SCHEME EXECUTION STAGES.....	- 92 -
FIGURE 4.36 CORTEX A35 PIPELINE STRUCTURE	- 94 -
FIGURE 4.37 CORTEX A53 PIPELINE STRUCTURE	- 94 -
FIGURE 4.38 CORTEX A57 PIPELINE STRUCTURE	- 95 -
FIGURE 4.39 IN KERNEL SWITCHER IN BIG.LITTLE	- 96 -
FIGURE 4.40 GLOBAL TASK SCHEDULING IN BIG.LITTLE.....	- 97 -
FIGURE 4.41 SIMULATION TOOL CHAIN	- 98 -
FIGURE 4.42 2 APP RESULT COMPARISON BETWEEN RE SCHEDULER AND IKS IN LINUX 3.14.....	- 99 -
FIGURE 4.43 2 APP RESULT COMPARISON BETWEEN RE SCHEDULER AND GTS IN LINUX 3.10	- 101 -
FIGURE 4.44 3 APP RESULT COMPARISON BETWEEN RE SCHEDULER AND IKS IN LINUX 3.14.....	- 103 -
FIGURE 4.45 APP RESULT COMPARISON BETWEEN RE SCHEDULER AND GTS IN LINUX 3.10	- 104 -
FIGURE 4.46 SIMULATED RESULT OF RE SCHEDULER IN S.M.B CPU CORE CLUSTER SETUP	- 106 -
FIGURE 4.47 COMPARING GTS AND RE SCHEDULER CPU UTILIZATION IN LINUX KERNEL 3.10.....	- 107 -

FIGURE 4.48 COMPARING IKS AND RE SCHEDULER CPU UTILIZATION IN LINUX KERNEL 3.14	107 -
FIGURE 5.1 SCHEDULING OVERHEAD AND PREDICTION ERROR OF RE SCHEDULER	111 -
FIGURE 5.2 RBM FORWARD LEARNING IN DPD IDENTIFICATION SUB LAYER	114 -
FIGURE 5.3 RBM RECONSTRUCTION DPD IDENTIFICATION SUB LAYER.....	116 -
FIGURE 5.4 DEEP BELIEF NETWORK.....	116 -
FIGURE 5.5 AUTOENCODER IN CLASSIFICATION LAYER	118 -
FIGURE 5.6 STACKED AUTOENCODER	119 -
FIGURE 5.7 RECURRENT NEURAL NETWORK UNFOLDED IN TIME.....	120 -
FIGURE 5.8 THE RE DFN OVERVIEW.....	122 -
FIGURE 5.9 FUNCTION BLOCKS OF LAYER 1	125 -
FIGURE 5.10 FUNCTION BLOCKS OF LAYER 2	126 -
FIGURE 5.11 PMP ENTRIES IN <i>AppSet Table</i>	127 -
FIGURE 5.12 DETAILED LAYOUT OF LAYER 3	128 -
FIGURE 5.13 PDP TABLE BIAS TERM ENTRIES.....	129 -
FIGURE 5.14 DETAILED LAYOUT OF LAYER 4	130 -
FIGURE 5.15 SCHEDULING SCHEME MAPPING BIAS TERM ENTRIES	130 -
FIGURE 5.16 DETAILED LAYOUT OF LAYER 5	131 -
FIGURE 5.17 SCHEDULING SCHEME PARAMETER MAPPING BIAS TERM ENTRIES.....	131 -
FIGURE 5.18 DETAILED LAYOUT OF LAYER 6	135 -
FIGURE 5.19 SIMPLIFIED LAYER 3 OF THE RE DFN BEFORE AND AFTER PRUNING.....	137 -
FIGURE 5.20 WEIGHT MATRIX IN SUB LAYERS OF RE DFN LAYER 3	138 -
FIGURE 5.21 SENSITIVITY MATRIX COMPUTATION IN SUB LAYERS OF RE DFN LAYER 3	138 -
FIGURE 5.22 PRUNING FACTOR DEDUCTION IN SUB LAYERS OF RE DFN LAYER 3.....	139 -
FIGURE 5.23 WEIGHT MATRIX Wl BEFORE AND AFTER CLUSTERING BASED WEIGHT GROUPING.....	140 -
FIGURE 5.24 SENSITIVITY MATRIX $Slme$ GROUPED ACCORDING TO WEIGHT GROUPS IN CLUSTERED Wl ..	141 -
FIGURE 5.25 CENTROID REGULARIZATION WITH SENSITIVITY CONSTRAINTS.....	141 -
FIGURE 5.26 SPARSE MATRIX STORAGE FORMAT COMPARISON ON WEIGHT MATRIX Wl	142 -
FIGURE 5.27 LAYER 1 TRAINING ERROR AND TESTING ERROR	145 -
FIGURE 5.28 LAYER 2 TRAINING ERROR AND TESTING ERROR	146 -
FIGURE 5.29 LAYER 3 TRAINING ERROR AND TESTING ERROR	147 -
FIGURE 5.30 LAYER 4 TRAINING ERROR AND TESTING ERROR	148 -
FIGURE 5.31 LAYER 5 TRAINING ERROR AND TESTING ERROR	149 -
FIGURE 5.32 LAYER 6 TRAINING ERROR AND TESTING ERROR	150 -
FIGURE 5.33 PRUNE FACTOR PLOT AGAINST TEST ACCURACY	151 -
FIGURE 5.34 PMP OVERHEAD REDUCTION IN INDIVIDUAL LAYERS AFTER OPTIMIZATION.....	152 -
FIGURE 5.35 COMPARISON OF OVERHEAD GENERATED OF RE SCHEDULER AND RE DFN	153 -
FIGURE 5.36 CPU ON TIME OBSERVED IN PMP VALUES FOR RE DFN IN LINUX 3.14 AND 3.10 KERNEL ...	154 -
FIGURE 6.1 GENETIC ALGORITHM CROSS OVER AND MUTATION	161 -
FIGURE 6.2 RESPONSE OF DIFFERENT ACTIVE FUNCTIONS	162 -

TABLE OF TABLES

TABLE 3.1 <i>AppSet Table</i> FOR QUIET INTERVAL SCENARIO	- 34 -
TABLE 3.2 THE PROBABILITY OF PRESENCE TABLE DERIVED BASE ON TABLE 3.1	- 37 -
TABLE 3.3 SEQUENCE TABLE DERIVED FROM TABLE 3.1 SORTED ACCORDING TO TIP.....	- 42 -
TABLE 3.4 <i>AppSets</i> COMBINATIONS SCHEDULING SEQUENCE REARRANGED ACCORDING TO TSP	- 42 -
TABLE 3.5 APP WEIGHTAGE TABLE FOR DIFFERENT VIRTUAL USERS	- 49 -
TABLE 3.6 THE SIMULATION RESULT	- 51 -
TABLE 4.1 PERFORMANCE METRIC PARAMETERS	- 59 -
TABLE 4.2 THRESHOLD OF CACHE MISS DOMINATING POWER DISSIPATION PROBLEMS	- 76 -
TABLE 4.3 THRESHOLD POWER DISSIPATION PROBLEMS RESULTED FROM CONTEXT SWITCH.....	- 77 -
TABLE 4.4 THRESHOLD POWER DISSIPATION PROBLEMS DUE TO SERIAL CODING BOTTLENECK	- 78 -
TABLE 4.5 GEM 5 SIMULATION PARAMETERS	- 95 -
TABLE 4.6 SUMMARY OF PDPs OCCURRED AND SCHEDULING SCHEMES APPLIED IN TEST SETS.....	- 105 -
TABLE 4.7 POWER DISSIPATION RESULT.....	- 108 -
TABLE 5.1 PERFORMANCE METRIC PARAMETERS	- 124 -
TABLE 5.2 HYPERPARAMETERS OF LAYER 1	- 125 -
TABLE 5.3 HYPERPARAMETERS OF LAYER 2	- 126 -
TABLE 5.4 HYPERPARAMETERS OF LAYER 3	- 128 -
TABLE 5.5 HYPERPARAMETERS OF LAYER 4	- 130 -
TABLE 5.6 HYPERPARAMETERS OF LAYER 5	- 131 -
TABLE 5.7 HYPERPARAMETERS OF LAYER 6	- 135 -
TABLE 5.8 HYPERPARAMETERS OF Re DFN AND THE AVERAGE ERROR RATE.....	- 144 -
TABLE 5.9 NEURAL NETWORK COMPRESSION PERCENTAGE AFTER OPTIMIZATION	- 151 -
TABLE 5.10 OVERALL POWER CONSUMPTION RESULT	- 155 -

TABLE OF ACRONYMS

Acronyms	Meaning
BP	Branch Prediction
CFS	Completely Fair Scheduler
DBN	Deep Belief Networks
dcache	Data Cache
DNN	Deep Neural Network
DSN	Deep Stacking Network
dTLB	Data TLB
DVFS	Dynamic Voltage Frequency Scaling
EAT	Effective Access Time
GA	Genetic Algorithm
GTS	Global Task Scheduler
IKS	In-Kernel Switcher
icache	Instruction cache
iTLB	Instruction TLB
MLP	Multilayer Perceptron
MMU	Memory Management Unit
NUMA	Non-Uniform Memory Access
OPP	Operating Performance Point
PCA	Principal Component Analysis
PDP	Power Dissipation Problem
PMP	Performance Metric Parameter
RBM	Restricted Boltzmann Machine
Re DFN	Re deep function network
RNN	Recurrent Neural Networks
SAE	Stacked Autoencoder
TIP	Thread Interference Power
TLB	Translation Lookaside Buffer
TSP	Thread Solitary Power
WSS	Working Set Size

Chapter 1 Introduction

1.1 Motivation and Objectives

The computing industry has been through many phases of revolutions since the beginning. It started with the demonstration adders built in Bell lab nearly 80 years ago and it took another 25 years to decommission the vacuum tubes and build computers as large as half a wall with transistors by IBM. 10 years later the hope of micro computing has been aroused with the introduction of integrated circuits, and the computing speed has been growing at an accelerated speed so fast, that no one by then has ever foreseen that it is going to hit the wall of its physics limitation 40 years later. The computing technology advancements are always breaking former boundaries to look for solutions to increase the computation power at a lower cost. To tackle the obstacles of today, multi-core / many-core architectures have been developed. However, just like other technology breakers before this, it does not guarantee a boost of computing power with no drawback.

The decreasing transistor size has created the power wall problem, which denotes the situation that switching on all cores in a processor to operate at high speed may cause the chip to burn due to the thermal dissipation. It is because of that transistors in the processors nowadays are fabricated with nanometer technologies and billions in numbers, and the amount of energy required to power them all together is too huge to sustain in a small chip area without causing overheating. In order to remedy the power wall problem, solutions like the dark silicon approach has been introduced [1] for processors with many cores. The dark silicon approach suggests to switch off some of the processor cores and operate the rest of the cores at full speed. When the operating cores are overheated, switch them off and rotate the tasks onto other cores that were switched off or in dark previously. Thus the processor is able to operate at full speed without worrying about the overheating problem as long as part of the processor is switched off or in dark. This approach extends the usability of the multi-core / many-core method. The multi-core architecture is originally invented to keep the processing speed advancement in line with the prediction of Moore's law. However, with a large portion of a processor chip in dark it is skeptical to say that the prediction of the law still holds.

If the power wall problem can be regarded as a disaster in the context of desktop computing and servers, then it is not much a literature exaggeration when addressing it as the apocalypse to the mobile computing platforms. Comparing to its stationary counterpart that has a seemingly unlimited power source to rely on, the mobile platforms are more power sensitive. The increasing screen size, screen definition, global positioning module, near field communication module, wireless communication module and graphic processing unit are all gaming for their own share of the energy stored in the lithium battery inside the devices. Besides that the software development is always in phase with the amount of computation power available. More visual appealing effects, more complicated new functions and complex structured multithread applications to process more data, regardless how much power efficient the processors can become the power required for application executions is always able to catch up in no time. In addition to all those energy related challenges mentioned above, it is rather impractical to adopt the dark silicon approach in mobile platforms, as the dark silicon approach requires additional chip area and higher temperature tolerance, which translate to one more thing that is lacking in a compact mobile computing platform, namely the space. Some may say, improvement in battery

architecture design and fabrication method can resolve the power hunger problem, but the Galaxy Note 7 incident has definitely taught the world a lesson that it is not a simple task and something can be rushed to at all.

Many solutions have been attempted to better the power issue. As addressed in various studies, the implementation of intuitive methods to manage the power dissipation in computing systems holds the key to overcome problems caused by excessive power dissipations generated in this era of dark silicon. One out of many solutions is power management schemes developed to reduce power dissipation in an operating system. Power management schemes can be utilized as an add-on or as a part of the kernel, such as a daemon or a scheduler.

The existing power management researches focusing mobile computing systems mainly exploit the parallelism of multi-core CPU with different power management schemes. For example the memory intensive application management scheme and the IO intensive application management scheme [1] are proposed to balance the workload according to different categories of applications. The application categories are predetermined based on fixed mappings generalized from statistical data collected. This kind of fixed mapping determined using offline analysis exposes newly incoming applications to the problem of not being accurately categorized to meet runtime requirement. Besides that, there are online management schemes that adaptively categorize the workload into different phases [2] and predict the workload based on the instantaneous reading of the system statistics [3]. The online schemes perform better in terms of quick response time and able to operate without conducting statistical analysis prior to deployment. However the accuracy of the categorization process is not as good comparing to the offline approaches.

Besides the power management schemes, there are also other power reduction techniques integrated as a part of the hardware or the operating system and they are able to be utilized by power management schemes via application programming interface (API) or system calls. For example, hardware based power saving mechanisms such as Power Gating that works towards the goal of reducing CPU power dissipation by switching off unused cores, and Dynamic Voltage Frequency Scaling (DVFS) [4] that reduces the power dissipation by lowering the operating frequency in CPU core executing less computation demanding processes. Such mechanisms can achieve remarkable power savings if the power management schemes are able to examine the power dissipation patterns of each application accurately. There is also software based scheduling techniques such as the smart thread migration [5], which migrates application threads from CPU cores with more workloads to core with less workload and maximize the power efficiency with the highest throughput at constant operating frequency. However, this technique is highly application dependent, and maximum throughput does not convert into maximum power efficiency automatically without considering the interference between different applications.

The importance of power management on mobile devices is rapidly increasing with the fast growing popularity of these devices and the applications (apps) running on them. There are millions of apps available in the Google Play (app store for the Android operating system) and Apple App Store, where different mobile device users may choose different apps based on their needs and personal preference. However an average mobile user frequently utilizes only 10 apps or less to fulfill the demand of their daily activities. For example, a businessman frequently accesses his mobile device to read documents and editing e-mails with e-mail app

and document viewer app, while a student may spend more time on social media and web browser apps. Since the combination of frequently utilized apps is unique and finite for each user, it is viable to analyze the combination and generate a pattern in app usage with a given user profile. Thus, it is assumed that by learning the app usage pattern of a mobile user, an accurately tailored power management scheme can be developed to optimize the different level of power dissipation of his or her device.

DVFS and heterogeneous system architecture are two pillars of the modern techniques in combating the power wall problem, where DVFS is able to control the utilization of CPU cores and heterogeneous CPU cores make allocating suitable computation resources for different workload possible. However, it is not a common practice to apply these two techniques based on high-level interaction between the user and the operating system. The app usage pattern can help to improve the efficiency of these two techniques and breaking the barrier between the user and the operating system, by providing information to the operating system on what the user is likely to do next and what is the computation impact likely to be. By incorporating these two techniques, the app usage pattern power management scheme is able to gain fine grained scheduling capabilities, such as reserving faster CPU cluster for app that requires more computation resources and slower cluster for app that requires to run a prolonged task at less computation intensity. Thus DVFS and heterogeneous system architecture are considered as the best supplement for the app usage pattern based power management scheme.

The app usage pattern based power management scheme aims to tailor an approach to mitigate power dissipation for each user differently, however the difference between mobile devices and app usage patterns has made such tailoring a complex problem to resolve without runtime analysis and tuning. Artificial Neural Network (ANN) is known for being able to solve universal approximation problems in an automated way with minimum human intervention, which is an ideal complement to tailor the power management schemes. An ANN aided app usage pattern based power management scheme satisfies most of the stipulation in the tailoring process, where app usage patterns can be automatically collected in the background at runtime and power dissipation can be optimized according to the patterns as the user uses the mobile device normally. Hence ANN is a technique that can be adopted to fully release the potential of the app usage pattern based power management scheme.

With the above motivations, this thesis aims to accomplish the following objectives. Firstly, develop a set of power management schemes to reduce the power and thermal dissipation in CPU cores. Secondly, improve the prediction of power dissipation caused by user applications and develop a model for power dissipation of individual applications with a monitoring structure at runtime. Thirdly, incorporate the scheduling schemes into a Linux kernel scheduler. Fourthly, deploy the scheduler in heterogeneous environments. Fifthly, automate the scheduler based power management scheme decision making routine with neural network algorithms.

1.2 Main Contributions

The thesis has three main contributions as described below with each key point highlighted in **Bold**.

First contribution, a **Thread Interference Power and Thread Solitary Power (TIP-TSP) model** is first proposed. The proposed model analyses the power dissipation relationship between different combinations of

application threads, which is named as **AppSets**. The amount of power dissipated by different *AppSets* are monitored and analyzed with the TIP-TSP model at runtime, which resembles an effective online power dissipation monitoring approach. The TIP-TSP Model [6] is developed based on the observation that in the same group of apps, the sum of power consumption of apps executed individually is less than the power dissipated by the apps executed concurrently. The excessive amount of the power dissipated amongst concurrently running apps is the TIP. The amount of TIP is unique to a set of application threads and the set is named as *AppSet*.

Based on the TIP-TSP model, several **user pattern aware scheduling schemes** are proposed. They are realized with **five scripts** running on top of the operating system to optimize scheduling decisions: The **monitor** script identifies the *AppSets* in a run queue and records their power dissipation. The **updater** script updates records in the **AppSet Table**, which keeps the *AppSets* and their power dissipation to aid the task of TIP-TSP component identification and calculates the TIP components based on the power dissipation monitored. The **optimizer** script decides the most power conservative scheduling sequence of a given *AppSet*. The **interleaver** script generates special scheduling sequence to help the identification of TIP component. The **scheduler** script decides which one of the **three scheduling schemes** proposed to apply with the given *AppSet* and optimized scheduling sequence.

The **three scheduling schemes** are: **HOLD**, **DEDICATED** and **Total parallelism**. The **HOLD** scheme reduces TIP dissipation by putting some application threads in the run queue on hold. The **DEDICATED** scheme reduces TIP dissipation by reserving the run queues for different CPU cores for different application threads and the **Total parallelism** scheme combines the function of the HOLD and DEDICATED schemes. Testing experiments are designed in a way to reassemble application usage of different mobile users in an Android environment on top of Linux kernel. In the experiments, there are three groups of simulated users. The **Business** user group uses mainly business apps. The **General** user group uses all kinds of apps and the **Media** user group uses media apps. The simulated results show that the proposed user app usage pattern based scheduling scheme manages to reduce TIP dissipation by up to 63% but manages TSP less efficiently. Thus the amount of power reduced by implementing the scheduling schemes is mainly from the TIP components.

Second contribution, **DVFS** is a technique for power reduction. It is incorporated with the proposed scheduling schemes in a scheduler named **Re Scheduler** developed in this work. In order to apply DVFS, **two quantization models** for accurate computation of power dissipation in the TIP-TSP model to make application aware scheduling decisions for Re Scheduler are developed. The prefix Re stands for the phrase “Recursive Enhancement”, which elaborates the behavior of the proposed Scheduler and DFN. The Re Scheduler and Re DFN optimize the system power dissipation based on the application information collected over time. The optimization efficiency increases as the information collected reproduces the characteristic of the application more closely. The information collection process is recursive and the optimization capability of the scheduler gets enhanced recursively at system runtime. Thus the prefix Re is used in the thesis for both the Scheduler and other scheduling mechanism proposed.

The first quantization model developed is the **Performance Metric Parameter (PMP) model**, which monitors the power dissipation of the system as a series of PMP event counts (cache, TLB, memory, branch miss, context switch, etc.) and converts the PMP counts into CPU cycles. The second quantization model is the

temperature and power dissipation model, where the recursive relationship between correlated operating **temperature**, **frequency**, **voltage** and **power** are studied and formulated. The PMP converted CPU cycles are used to calculate the **sampling period**, which is used to compute **energy budget** with power figure calculated using the power dissipation model. The energy budget represents the amount of power saved by the Re Scheduler, and the Re Scheduler is able to boost the performance of certain CPU cores using DVFS as long as the energy consumption is within the budget.

To select the suitable scheduling scheme for a given power dissipation pattern of an *AppSet*, a group of **Power Dissipation Problems (PDPs)** are identified and analyzed. Each PDP is then mapped to a set of scheduling schemes selected, and with the help of PDP Mapping, the complexity of scheduling schemes selections and setting the scheduling parameters is reduced. There are **nine different PDPs**, and nine sets of **PMP threshold** values are used to identify the PDP problems by comparing the PMP values monitored and the threshold values. Three main root causes of power dissipation in a computing system have been considered during the process of PDP identification, specifically **cache miss**, **context switch** and **serialized coding bottleneck**. A miss in the cache is capable of incurring up to 2 magnitudes of latency in main-memory access [7] and a partial cause of additional cache prefetch miss [8]. Context switches can pollute cache lines [10], generate unnecessary cache upgrades [11] and inflict additional overheads to the system [12]. Serialized coding bottlenecks are present less paralleled multithreading application threads, which is a likely cause of CPU pipeline halt [13].

The proposed Re Scheduler is constructed with multiple function blocks, namely *AppSet Table*, Cost Table, Table Walker, Core Walker and thread prediction. By going through different blocks, the amount of power dissipation reduced by implementing the scheduling schemes can be determined. The amount of power dissipation reduction is updated using the energy budget conversion, and reserved for Re Scheduler to boost system performance. There are **three** schemes used in Re Scheduler: **HOLD**, **DEDICATED** and **SUPER**, with the total parallelism scheme developed earlier modified and renamed as SUPER. The efficiency of Re Scheduler is tested in a system constructed with **three CPU clusters**, and compared with the native **Global Task Scheduler (GTS)** and the **In Kernel Switcher (IKS)** Linux schedulers. The results show an improvement of 8.01% to 22.32% in terms of power reduction. However, various problems such as excessive computation overhead, race condition due to limited PDP to PMP mapping options and scheduling scheme inflexibility were identified from the results and analyzed.

To resolve problems identified in the analysis of the results from the Re Scheduler, **neural network based methods** were explored. Multiple **neural network algorithms**, such as the RBM, Autoencoder, RNN and DBN, were adopted. It is observed that each algorithm is able to handle a different category of problems at maximum efficiency, and this encourages the design of the **Re Deep Function Network (DFN)**.

Third contribution, the Re DFN is constructed using multiple neural network algorithms to form a deep neural network with multiple layers, aiming to construct Re DFN for solving the nonparametric PDP to PMP mapping and scheduling scheme picking problem with DNN structure and yet keeping its computation overhead low. Each layer of the Re DFN is loosely coupled and trained individually. There are **six layers** in the Re DFN and each resembles the function of Re Scheduler partially. **Layer 1** extracts the feature from PMP monitored. **Layer 2** selects the features used to take part in the PMP to PDP mapping. **Layer 3** conducts the mapping

between the PMP and PDP. **Layer 4** maps the selected PDP to available scheduling schemes. **Layer 5** sets the parameter of each selected scheduling scheme. **Layer 6** compares the projected PMP with the monitored PMP and update the DFN with backpropagation algorithm.

The initial test of Re DFN shows that it is able to achieve **better prediction error** with **more computation overhead** as comparing to the Re Scheduler. In order to reduce the computation overhead, **neural network optimization** methods such as neural network pruning, weight sharing and matrix compression have been applied.

The optimized Re DFN has shown an improvement in computation overhead up to 80% compared to the Re DFN without optimization and a nearly 50% of computation overhead reduction compared to the Re Scheduler. The experiment set up is identical to what has been done for Re Scheduler and it is shown that the Re DFN is able to achieve a better power dissipation reduction of 4.5% on average. The Re DFN expands the horizon of making complex system optimization at runtime in dedicated neural network engines without impacting the system performance.

1.3 Organization of the Thesis

The thesis is organized as follows. Chapter 1 presents the motivation for this work and defines the work objectives. It also summarizes the contributions of the work. Chapter 2 reviews the related existing works, including multi-core parallelism, scheduler based power management, system performance quantization, system power dissipation, heterogeneous architectures and neural network aided time series forecasting. Chapter 3 presents the proposed TIP-TSP model and the development of the proposed scheduling schemes as well as its simulation results. Chapter 4 presents the TIP-TSP quantization methods and power dissipation problems (PDPs) to scheduling scheme mapping in the proposed Re Scheduler and the simulation results. Chapter 5 improves the performance of the Re Scheduler and demonstrates the construction of the neural network aided Re DFN and the evaluation of its performance. Chapter 6 concludes the thesis with discussions and future works.

Chapter 2 Literature Review

To tackle the power dissipation problem in mobile computing platform various areas of studies have been explored. This chapter addresses some of the literatures studied in the development of the work proposed in this thesis. The first two sections, 2.1 parallelism and multi-core processors and 2.2 power management approaches are covering works studied to build the user app usage pattern based scheduling schemes which are going to be introduced in Chapter 3. The next three sections, 2.3 system performance metric quantization, 2.4 CPU temperature and power dissipation and 2.5 power management in heterogeneous architectures cover the related studies that help to develop the power dissipation quantization models and heterogeneous system focused Linux scheduler in Chapter 4. The last sections, 2.6 neural network application and optimization introduces the fundamental studies reviewed to build the neural network based Linux scheduler that is presenting in Chapter 5. By studying existing power optimization techniques in the Literature Review section, there are indeed several dynamic power management approaches tend to solve thermal and power dissipation simultaneously. However the approaches reviewed either required offline training prior to execution or not built for complex scheduling scenarios in multithreading environments. Unlike the existing works, the Re Scheduler and Re DFN resolve multithreaded scheduling problem at system runtime with power and thermal awareness.

2.1 Parallelism and Multi-core Processors

Parallelism computing speeds up the problem solving process by dividing large scale problems into smaller parts and executing them simultaneously. The realization of parallelism involves the processor with multi-core architecture. The divided problems could only be executed simultaneously with the presence of multiple cores in a processor. Ideally the execution time is inversely proportional to the number of cores. Meanwhile the size of the transistors shrinks to build processors with less area and less power dissipated. The dark silicon effect had ended the simple approach to optimize the computation speed and power dissipated at the same time. The physical limitation of atoms had brought the massive amount of thermal dissipation in processor cores. The leakage power presented in billions of transistors had made it impossible to turn on all the core at once in large scale many-core designs. To solve the dark silicon problem, the term power management had been emphasized more than ever. Various solutions had been studied across different disciplines. Heterogeneous designed processors turned out to be the better solution in solving the dark silicon problem and maintaining the computation power possessed with homogeneous designed processors. To show the performance comparison between a well-managed heterogeneous system and a symmetrical general purposed system, in [9] Allred et al. had discussed the possibility of overcoming the dark silicon limitation in symmetrical architected multi-core processors and proposed a DSA (Dark Silicon Aware) system to improve the energy efficiency by 5.7 times. The drawback in the system is that the CPU cores are dynamically assigned with different operating frequency

according to their dark silicon effect levels. This assigning process is capable of producing overheads under heavy workload when frequent change of CPU core frequencies occurs.

One of the reasons that the heterogeneous designs outperform the homogeneous designs in specific scenarios is that the asymmetrical architected processors always consist of chips for dedicated purpose operations. In [10] Hao et al. had discussed the heterogeneous multi-core architecture from the DSP perspective. In the findings the increasing complexity of DSP (Digital signal processing) applications resulted in massive amount of system resource harvesting in the conventional general purpose CPUs. This symptom created race conditions between applications scheduled based on parallelism computing. Hence a dedicated programmable stream controller was proposed in the paper to forcefully assign the essential streaming tasks to the DSP cores. In the result, the proposed stream controller demonstrated 80% performance improvement [10] compared to the software based DSP controller operating on the homogeneous main processing unit. The drawback of this approach is that a dedicated DSP controller must be designed individually and fabricated with the main processing unit, which is not cost effective to adopt in multi-core designs with less DSP demands. Optimizing the performance of asymmetrical designed processors requires different approaches compared to the symmetrical designed processors. In the asymmetrical designs the similar optimizing approach cannot be applied to all processing blocks as they are different. In [11] Li et al. had explored the feasibility of genetic algorithm in a heterogeneous multi-core architecture with NoC (Network-on-Chip) Communication structures. The genetic algorithm dynamically performs local optimization of software and hardware partitioning in the heterogeneous system. The result has shown that the algorithm proposed is capable of achieving near to the global optima for hardware and software partitioning in a SoC (System-on-Chip) without achieving optima at each computation node. The implemented algorithm is not capable of improving the global optimal partitioning efficiency. Even though the heterogeneous designs had shown better performance in some operations compared to the homogeneous general purpose processing system, there are other approaches studied to reduce the dark silicon effect while maintaining the homogeneous architecture. In [12] Raghunatha et al. had studied task scheduling priority based on homogeneous design regardless of the dark silicon effect. The experiment is based on Snipe simulator, where the process variations in identical cores were taken into account. The algorithm identified the best performing cores and then “Cherry-Pick” them to perform task with higher priority, i.e. to assign the better performing cores with higher frequency and voltage pairs. The experimental result has shown 30% performance improvement with half of the processor cores in dark. The problem is that the simulation was conducted with only one application. Thus there is the possibility of performance degradation in a multithreading scenario.

The studies discussed earlier in this section had shown the possibility of achieving heterogeneous managing approach in homogeneous systems. This shows that although the heterogeneous solutions are efficient in dealing with specific known scenarios, the presence of homogeneous processing units are still crucial to maintain the ability to solve general computing problems with instantaneous decisions. The review of the current multi-core parallelism studies had inspired the proposed power management scheme with a few points, which include the importance of notifying popular operations in a system and designing dedicated execution approach to reduce the load on general purpose units, and the importance of considering adapting different multithreaded workloads with different management schemes.

2.2 Power Management Approaches

The state-of-art dynamic power managers manage the system power dissipation with two approaches, the dynamic voltage frequency scaling (DVFS) and the scheduler oriented. DVFS oriented approach changes the operating frequency and voltage of CPU cores to conserve power dissipated over time. The scheduler oriented approach allocates system resources for application threads. The existing methods studied can be categorized into three categories, the DVFS based power management methods, the scheduler based management methods and the hybrid methods, i.e. the power management methods incorporated with the DVFS and the scheduler. The features of prominent methods of each category are discussed in the remaining part of this section.

By reviewing the DVFS based methods, the limitation of the method implementation comparing to the scheduler methods is observed. In most of the commercial symmetrical processors, the same frequency and voltage pairs are allocated to all the CPU cores. As a result DVFS based power management schemes show marginal performance improvement with large computation overheads. In [13] Almoosa et al. presented a DVFS based adaptive power budget controller. The algorithm of the adaptive controller was designed to dynamically manage the power dissipation with no initial statistic base. The initial power dissipation was profiled against an asymmetric multi-core processor. A controller was assigned to each core with a dedicated power budget. Each controller decides for the core the amount of the power budget used. The result shows that the settling time of changing power budget is 15 ms or 50% faster in the simulations with the adaptive controller compared to the simulations without the adaptive controller. The problem with the proposed controller is that without a statistic base the training time of the controller is long. The real-time tracing mechanism utilized to profile the power dissipation took a significant amount of system resource.

To achieve heterogeneous like voltage and frequency scalability on a homogeneous system, Chakraborty et al. had acknowledged the limitation of DVFS due to dark silicon effects and hence proposed a heterogeneous power-performance optimization method for homogeneous systems in [14]. They demonstrated a system level management algorithm to dynamically group identical cores. Different voltage-frequency pairs were issued to the processor core groups. Hence a heterogeneous system simulating on a homogeneous architecture was realized. From the experiment simulated, it shows a 22% improvement in energy efficiency. The drawback of this method is the lacking of a runtime adaptation module for the DVFS scheme. Thus it did not manage to reduce the runtime overhead. In [15] the clustering DVFS management scheme had been discussed. The CPU cores in a processor will be dynamically grouped into different clusters based on the deadlines of their threads assigned. The clusters with threads of similar priority and deadline will be given the same frequency and voltage pair. Hence the tasks can be paralleled regardless of the load balancing issue. The result shows that the power dissipation can be reduced up to 15%, but the performance will be degraded by 10% as a tradeoff. Other than adjusting clock frequency and voltage of the processor cores, there is the DVFS based approach that increases system performance by gating the unused the processors cores named power gating. Power gating is capable of

forcing the CPU to run in energy saver mode while performing non calculation demanding operations. In [16] the power gating method was realized by structuring a clock tree to reduce the clock skew upon clock gating commands. It is capable of reducing an average of 8% of clock idling power. However the structuring of the clock tree requires additional flip-flops that cause the chip area to increase by 6%. In [17] the power gating method was adopted with power capping CPU utilization at various virtual machines. The power capping method gates the CPU cores of each virtual machine and migrates excessive workloads to less occupied virtual machines. As shown in the result the power dissipation at single CPU can be reduced up to 80%, but the overhead size and the migrating time cost were not examined in the work.

Power management methods of the scheduler category had shown their flexibilities with numbers of variation in the existing studies. In thermal aware scheduler methods, the scheduler dispatches the processes to less occupied cores and migrates the process from cores with larger thermal hot-spots. In [18] the scheduler that schedules application threads according to a power thermal model is proposed. A thermal map of the CPU cores is generated dynamically at runtime and the power dissipation of application threads is calculated based on the thermal map. The application threads are scheduled to the cores with less thermal power. The proposed method is able to achieve 30% improvement in processing speed on processes with less than 6 seconds of execution time, but for the processes with longer execution time the improvement is only 4%. In [19] the thermal aware scheduling was adopted with the implementation of a lookup table. The lookup table contains the schedule ability of each core. The tasks waiting for core assignments are allocated to the core with higher schedule ability. The running tasks in the cores with lower schedule abilities will be migrated to cores with higher schedule abilities. If the cores were still overheated after migration, they will be issued a Turn-off command to cool down. The result shows that the temperature of cores can be maintained under the threshold temperature in 83% of the time, but the performance will degrade by 4.7%. In [20] a low complexity embedded system thermal scheduling scheme with extreme power constraint had been proposed. The proposed scheduling scheme monitors periodical tasks and schedules the periodic tasks according to the power budget computed for each thermally monitored CPU core. The power budgets are computed based on the energy that the system can harvest through a solar panel. The result has shown that with periodical tasks well predicted the system can harvest enough energy to power up to 3 cores of the CPU, but the complexity of the periodical tasks must be kept low.

There is always a notable amount of performance and energy trade off by applying power management schemes. The optimization aware scheduler methods are adopted to overcome this problem. In [21] a negative power savings aware power management scheme was proposed. The CPU cores will be issued with idling command to save power upon the absence of incoming tasks. The incoming task arrival sequence is expected to be a Poisson distributed series. The proposed scheme is able to improve the CPU utilization by 16% with given power budget. But in the case where the task arrival sequence is unpredictable, the utilization will degrade. In [22] a throttling based DTM (Dynamic Thermal Manager) was proposed. The proposed manager focuses on optimizing CPU thermal monitoring window to regulate the delays occurred in the triggering phase of the thermal sensing and policy response. To reduce the delays the DTM is distributed into three blocks, namely the fetch throttling, the rename throttling and the register file occupancy throttling. Each DTM block will monitor and make throttling decisions based on events occurred in their relevant pipelining stages. As discussed in the result in [27], the DTM is capable of regulating thermal hot-spots with 400 μ W of overheads produced. However the authors also admitted that there were a lot of possible power dissipation modules not modeled in the

simulation. In [23] the optimization on DSC (Distributed Shared Cache) had been discussed. The proposed method increases the locality of DSC and hence reduces the power dissipated related to DSC referencing and memory accessing. The simulated result shows averagely 55% of DSC targeted at local L2 (level 2) cache. But the result cannot reflect the actual system performance without the presence of the virtual memory pages assumed in the study. In [24] a deadline scheduler for Linux based mobile system was proposed. This scheduler is part of the resource management frame work named ACTOR and has produced significant improvement in runtime application multithreading. As addressed in the result, the scheduler is able to multiplex DSP intensive applications at 80% throughput increment in comparison with the native Linux Scheduler, but for non DSP intensive applications only marginal performance improvement was achieved.

In the following studies the scheduler was modified to implement prediction based power management. The power management predicts the future workload based on the system performance. In such case the power dissipation and response time are reduced. Different predicting methods were proposed by the researchers. In [25] a low level self-calibrating power manager was proposed. The current processing parameters at current interval are deduced based on the system values stored in the historical lookup table. The lookup table is continuously updated upon processor requests. Thus the runtime processor parameters will be adjusted over time to suite the slow aging cores. The managing module proposed is capable of making relevant processing adjustment to time averaged changes, but cannot respond to great deviation in system performance. In [26] a periodic power phase predictor was introduced. The predictor proposed predicts the power level at next period based on a global power phase table. The table records a power phase with relating parameters such as its length, level and time stamp. Tasks will be allocated according to the CPU core with lower power level predicted. The result shows that the proposed predictor is capable of reducing the power dissipation by 5.4% but also increase the performance by 3.8% comparing to the reactive power manager in Windows Vista. But the proposed method had only demonstrated its effectiveness against single threaded tasks. The accuracy of the predictor under multithreading scenarios is not considered.

The management methods with scheduler and DVFS incorporated are capable of managing the power from multiple levels in a computing system. A few studies with intuitive incorporated power managing approaches are discussed in the following. In [27] the implementation of application slack identification had demonstrated the possibility of workload relocation. A slack is an interval occupied by a running thread with CPU resource unused. The slacks are identified by the proposed scheduling method and removed from the queue to save CPU time from being wasted. The removed slacks are later added to the back of scheduled threads to avoid error caused application convergence. The DVFS controller will assign lower frequency and voltage pairs to the CPU cores to process the slacks scheduled. The result shows a 33% of energy reduction with a H.264 application. Again the performance improvement of multithreaded scenarios was not demonstrated in the research. In [28] the proposed EDP (Energy Delay Product) based optimization method was realized with DVFS and scheduler combined. The EDP values of each application are calculated based on their runtime power dissipation and execution time. DVFS will be rotated for each application and the EDPs of each application will be recorded. The optimized DVFS and EDP pair will be logged into the DVFS table to save runtime computation overhead. The result has shown overall energy reduction in comparison with performance biased scheme. The drawback of the research is lacking of a prediction module. In [29] the power aware DPM (Dynamic Power Manager) PCOUNT was proposed. PCOUNT fetches threads according to their priorities at runtime. The priority of a

thread is evaluated based on the requested memory and computation sizes. The memory and computation modules in the computing system are assumed to dissipate more power as the sharing threads increase. Hence threads requesting more system resources will be given lower priorities and wait for scheduling in an interval with less resource racing threads. The proposed DPM proves the power efficiency over existing schemes by 26%, but the proposed DPM did not provide measures to resolve inter thread dependencies. In [30] the SFAA (Single-clock domain multi-processor Frequency Assignment Algorithm) which determines the global operating frequency with task period consideration was proposed. The lowest possible operating frequency for a thread is determined based on possible intervals in between the preemption from the tasks with higher priorities. The frequency will be issued to the scheduled thread to complete at the interval between two preemptions. The SFAA schedules threads based on their evaluated frequencies and the utilization. Threads issued with similar frequencies and operating intervals will be scheduled into the CPU cores according to their utilization. In the result SFAA shows a 55% power saving compared to existing scheduling schemes in octa-core systems. But as addressed by the authors, the actual performance of the SFAA will drop due to the continuous nature of this frequency calculation, where the discrete DVFS pairs can cause idling states for threads with non-ideal operating intervals. In [31] a statistical CPU power prediction model for virtual CPU DPM was proposed. The proposed power model was achieved by training an existing set of power models with only CPU intensive training sets. After proving the accuracy of models they were testified with IO, memory and CPU intensive workloads. The result of rectification was indexed to estimate the power dissipation according to application categories. The result had shown that the models can achieve 90% accuracy in prediction of CPU core and frequency combinations. The restriction of this method is that the selectiveness of training set prolongs the training cycle of the models. As shown above, the Scheduler and DVFS incorporated management methods are capable of complying the native process deadline and optimizing the performance at the same time.

To increase the accuracy of prediction and modeling based power management schemes a well-established learning structure is crucial. In [32] [33] a Bayesian Classifier based SL (Supervised Learning) DPM scheme was introduced. To realize the proposed SL approach, at first a training set is created which consists the input features (queuing time and arrival frequency) and output measures (power dissipation and execution time). Later the training set is used to train the classifier that can predict the output measures based on runtime input features. To improve the accuracy of the trained classifier the boundaries of predicted outputs have to coincide with performance boundaries. Upon accurate output power prediction the power management policies are assigned to reduce the idling power dissipation in less occupied CPU cores. The result of the proposed DPM shows an 8% power saving comparing to an existing feedback-control loop based method and 18% saving comparing to an existing offline power managing method. The drawback of this work is that the instantaneous output measure based prediction correction method limits the prediction window to a short period of time. In [34] a task aware scheduler was proposed. Each task is given a time model consisting of the running time requested (rt) and the actual execution time (at). A task will be given higher priority if the at value is close to the rt value. The tasks in each processor core are monitored and core availability levels are indicated based on the given power budget. The scheduler will schedule tasks according to their priorities and the availability of the processor cores. In the cases where the core availabilities are equal, the task will be assigned to the core with the most numbers of successfully completed tasks. The scheme was tested in a large distributed computing

system. The result shows an average 20% of performance gain comparing to the native scheduling schemes. Being a scheduler lack of system parameter learning capabilities, the proposed scheme cannot adapt to the heterogeneous systems. As a result the performance gain of symmetrical systems is similar to the gain of asymmetrical systems. In [35] the online learning system based DPM and DVFS method was proposed. The task execution time in the proposed method is decomposed into two parts, the CPU execution time and the cache missing time. The DVFS controller will set lower frequency and voltage pairs for cache missing intervals of an executed task. The DPM will schedule the task according to the decomposed execution time with DVFS controller synchronized. The controllers adopted in the proposed method are trained with online-learning approach where the task traces were learnt one instance at a time. The result shows a 49% of energy savings. Due to the limitation of online machine learning approach, the accuracy of the controller will drop as the number of the instance to predict increases. In [36] a workload profile based power model was proposed. The workload defined in the proposed model is a combination of applications where each application is described as a sequence of tasks. For example upon scheduling the SMS (Short Message Service) application will perform the processing interrupt, display and sound, key-handler, display, key-handler and finish tasks in sequence. Each of the tasks was trained with runtime system parameters to produce a combination of power profile. Users having different applications usages are categorized into different profiles. In the result, the model has shown accurate power predicted for different user profile. The drawback of the model is lacking of consideration on user interactive apps. In [37] a mobile usage pattern analysis had been presented. The mobile usage patterns discussed in the analysis were only limited to the main mobile system modules such as the WiFi, 3G and Call. The analysis was conducted by collecting the usage patterns and the mobile system power dissipated of several users over a period of time. The result has clearly shown the relationship between user usage patterns and the power dissipation, but the relationship of user interactive app usage pattern and the power profile was left undiscussed.

The reviewed studies had inspired this management scheme to adopt Scheduling managing approach and to incorporate with DVFS strategies. The system model of the proposed management scheme will cover the points that had not been addressed by the existing studies which includes the multithreading situation consideration and the app usage pattern recognition.

2.3 System Performance Metric Quantization

Monitoring and analyzing the power dissipation of a specific application is important for making application usage aware scheduling decisions. It is not an easy function to realize at CPU runtime, as the power dissipated by a computing platform is a composition of all applications executed concurrently and the power dissipated by each of the applications is an aggregation of a tremendous amount of operations taking place in different memory and execution units of a processor. Many researchers have studied the relationship between the power dissipation and the access pattern of the memory and processing units in a processor. The relevant studies that help to develop the power dissipation analysis of individual applications in a multithreading environment are addressed in this section.

In [38] Molka et al. have studied the energy consumption pattern of data transfers and arithmetic operations in Intel and AMD x84-64 processors. The study demonstrates an intuitive approach to access the contribution of each individual units in a process such as caches, buffers, arithmetic logic units and floating point units. An assembly coded workload has been developed to stress processor units in each core individually by pinning the workload to a dedicated CPU core in a round robin manner with the help of `sche_setaffinity()` function call. The overall system power measurements are collected with a dedicated external power meter and the power measurement patterns are studied to reveal the power consumption of each individual components. The results show that the power consumed by the execution of an instruction varies depending on the length of pipeline stages involved in the operation and the location of the data stored. The instruction that needs longer pipeline stage to compute such as the floating-point unit (FPU) dissipates more power compare to the Load/Store operation and accessing data in RAM dissipates the most power compare to caches. The study provides a good theory fundamental to the proposed performance and power quantization approach that the power dissipated through different processor units should differ according to their access pattern and can be computed if the number of accesses to each processor unit is known. However it is not as portable as the proposed approach that is able to generalize the power dissipation of individual application without external power meter monitoring mechanism.

To quantize the power dissipated by each processor unit at instruction execution stages, relevant research works that focus on the impact of each individual process units on the total CPU performance and their specified access times are studied. In [39] Kamble et al. have developed an approach to model power dissipation in caches using counts for hits, misses as well as numbers of reads and writes. The cache model is constructed with the consideration of power dissipated basic components of set-associative caches, such as the data address line, tag comparators and tag lines. The model has shown an average 15% of over prediction of power consumption, which is viable to serve as a worst case power dissipation power consideration, but the inaccuracy is likely to be reflected as additional overhead for a prediction dependent system. In [40] Kumar et al. address the growing interests of incorporating accelerators to speed up the specific operations in processors requires better cache coherence to accommodate the data transfer between different cache hierarchy of accelerators and the processor. A light weighted coherent cache hierarchy known as FUSION has been developed which is able to gain 4 times of performance improvement. In the development of FUSION, the characteristics of SCRATCH architecture have been studied, and the SCRATCH modeling approach has been considered to generalize the cache coherence related power dissipation problems in the proposed study.

In [41] key characteristics of different cache designs have been addressed, and the pattern of cache miss rate variation according to different working set sizes has been notified. Although the cache miss rate changes according to different cache designs, the size of the working set is always the prominent factor that causes misses in cache access, which has encouraged the proposed study to consider the relationship between the quantified cost of a cache miss and the size of the working set. In [42] Chen et al. address the cache management issues in energy efficient GPU computing. One of the issues in discussion demonstrates that the L1 cache miss rate is likely to be nearly 90% as the benchmarks get heavily executed in parallel with very large working sets. The L2 cache is experiencing the same problem and shows similar miss rate increment once the working set size gets larger than the size of the cache. The above mentioned works have helped the proposed work in this thesis in acknowledging the relationship between working set size and caches. However there are more factors

to consider other than working set size and cache hierarchy in order to deduce accurate cache miss cost. In [43] Puzak et al. discuss gauging the performance impact of cache misses using the time delayed in the pipeline. Other than cache miss it is discovered that the pipeline delayed time can also be utilized to analyze branch and prefetching misses. The method developed is named as the pipeline spectrum, where the impact of the miss events are quantized using cycles per instruction. However this method is not really capable of determining the cost in terms of CPU cycles in a cache hit event as proposed in the thesis, and the overall power dissipation is an aggregation of the miss and hit events. Thus studying the cost of hit event is essential to determine the total power dissipated at runtime. In [44] the impact of cache misses on the power dissipation in a Dynamic Voltage Frequency Scaling (DVFS) environment has been addressed. In DVFS the operating frequency and voltage changes according to the Operating Performance Point (OPP) picked, which results in different power dissipation as CPU power is a function that contains variables of frequency and voltage. In heterogeneous multi-core architectures the OPPs of different cores are likely to be different, hence it is crucial to develop a quantization approach that can be easily adopted to compute the impact of the same miss or hit event regardless of the OPPs.

Other than cache access events excessive power dissipation can be generated in other stages while an application gets executed through a CPU pipeline. In [45] Eyerman et al. and Gandhi et al. in [46] have studied the effect of branch miss in detail and found that miss rate is highly correlated to frontend pipeline length, the number of instructions executed since the last misprediction and the amount of L1 data cache misses. Although the studies do not provide a general model to quantify branch miss, they indeed illustrate the relationship between the caches and possible impact of a miss prediction in terms of pipeline stage flushes. In [47] Kadjo et al. have proposed a data prefetcher driven by branch prediction and effective address value speculation. The prefetcher predicts the future trend of the incoming instruction branches based on the historical register transformation states. The prefetcher is validated in a cycle accurate simulator and shows a geometric mean speedup of 20%, which indicates that the cost of a branch miss prediction can be bounded by the accuracy of ability of the branch predictor. The existence of translation lookaside buffer (TLB) reduces the round trip latency for applications to allocate remote virtual memories, which is a rather common latency component that impacts the overall system performance. The TLB miss time is normally bounded by multiples of the RAM access time depends on the structure of the Table Walk. Hence it is one of the focuses of the proposed quantization in this thesis.

In [48] Jayneel et al. have discussed the possibility of converting a hardware assisted page walk into software assisted page walk by intentionally corrupting page table entries. The framework mentioned can be utilized to monitor the system performance abnormalities and identify responsible applications. However the framework generates an unstable latency that ranges from 2 to 40 times depending on the TLB miss rate. The quantification approach proposed in the thesis is able to achieve better TLB cost deduction with a much lower latency. In [49] Vivek et al. introduce a virtual memory management enhancement approach to address the problem of page table overhead. In the current fine grained memory management technique it is common to adapt the memory page to the needs of different applications by reducing the page size. However as smaller page sizes stress the TLB and results in higher page walking latency, the number of TLB count and the page size of the targeting application thread are both considered in the proposed quantification approach.

Context switching is noted as one of the operations that costs the most numbers of CPU cycles. A context switch does not only impact the system performance with the migration cost that involves moving application threads from one core to another through multiple layers of memory units, it also generates additional cache misses, TLB misses and scheduling delay due to scheduling epoch misses. In [50] Li et al. have categorized the cost of a context switch into two types, the direct cost and indirect cost. The direct cost can be easily quantified, while the indirect cost is a result of cache sharing of switched application threads and it is difficult to be monitored. In [51] David et al. have conducted the context switching cost study specifically in ARM platform. It is noted in the result that the context switching time takes up to about 1 percent of the total CPU cycles elapsed. However as indicated by the authors, it is likely that the indirect cost is not fully reflected as the test case does not perturb the TLB and pollutes the caches. Another reason of the extremely small context switch overhead is that all the application threads are given 70ms of time slices, which is considered as very large with reference to the default Linux Round-robin scheduling.

In [52] Liu et al. dedicatedly conduct the study on cache miss related to the indirect cost. The result has shown that upon a context switch the efficiency of the prefetcher is likely to drop by 33% and the amount of cache misses can increase up to 10% in best cases and all the way till 300% in the worst case benchmarks. To avoid the complicated dynamic in separating the indirect context switching costs from direct context switching cost, in the quantifying approach proposed in this thesis cache miss events are analyzed separately from the direct cost of context switch.

As discussed above, in the situation of handling workload with large working set size is likely to generate cache misses and increase the memory access time to be equivalent to the time taken to access DRAM. The time taken to access DRAM is long and dreadful, but the actual access time varies according to the distance between the processor and targeting the memory node in the modern Non-Uniform Memory Access (NUMA) architecture. To accurately gauge the effect of NUMA distance on the memory access time the following research works have been studied. In [53] Hollowell et al. have discussed on the effect of manual NUMA tuning in Linux environment making use of the `numactl` and `nomad`. The tuning result shows that tailoring NUMA adjustment to individual application threads with `cpuset` is more effective than using `sched_setaffinity()` and `mbind()`, and it is able to achieve notable amount of performance boost in a native NUMA aware scheduling environment.

2.4 CPU Temperature and Power Dissipation

The prerequisite for a scheduler to make power dissipation aware scheduling decisions is an accurate CPU power dissipation model. The CPU power dissipation consists of two components, the leakage power (sometimes also known as the static power) and the dynamic power. The dynamic power dissipates through the switching activated logic gates and the leakage power dissipates to the leakage current. The dynamic power is frequency and voltage dependent, where higher frequency indicates that the switching of a logic gate is more frequent hence the power dissipation will be higher. The leakage power increases as the temperature increases, hence the three main factors, namely the operating frequency, the operating voltage and the temperature, are

considered in order to determine accurate power dissipation of a CPU core. In [54] Qi et al. have discussed an attempt to construct a temperature to power mapping method with considerations about different power models, process variations and leakage estimation. However it shows that utilizing power to temperature mapping has better efficiency in determining the power dissipation pattern. In order to derive accurate power dissipation figures based on the given thermal hotspots monitored in the processor, the complexity of the mapping problem has been reduced with thermal map-filtering and suppressed the noise terms with the least square optimization. The result has been tested in both the steady state where the temperature stays constant and the transient state where the temperature dissipates and increases according to the ambience temperature and the power dissipated over time. Due to the complexity of the transient state modeling, the result is not as promising as what has been achieved in the steady state. In [55] Jayaseelan et al. have developed a temperature aware task sequencing mechanism to compliment the function of Dynamic Voltage Frequency Scaling (DVFS). The system is targeted to resolve periodic heterogeneous tasks under timing constraints. The first step to optimize a given task sequence is to determine its maximum steady state temperature. Iteratively the historical power dissipation pattern of each member task of the sequence is examined with a heuristic search. The task sequence that generates the lowest thermal energy is then chosen as the optimized task sequence. The result demonstrates a 10°C drop or 10% of thermal energy dissipation reduction, which is achieved with the per task level accurate transient state thermal model adopted in the study.

The abstracted power dissipation and thermal transient models do provide a simplified approach to deal with prediction problems within a certain die area without compromising the accuracy. While the detailed full system studies unveil the thermal dissipation problem with a different perspective. In [56] Skadron et al., a team from the University of Virginia that has developed the HotSpot thermal model, discuss in detail about the thermal resistance and capacitance base thermal model used in the HotSpot [57]. The thermal model constructed is tested and verified with finite element method. The model is later applied in various dynamic scheduling techniques to provide an additional degree of thermal awareness in the existing mechanisms. For example the lower voltage value is always chosen as the OPP for the processor to be thermal conservative. Since a given voltage is likely to accommodate multiple levels of operating frequency, being thermal conservative is one of the better options that brings more power efficiency without compromising much performance. Despite the accuracy of the temperature model, the actual performance of the prediction is bounded by the overhead generated by the on die temperature sensor locations. Hence regardless of the model complexity, a good amount of assumption is always required in practice. In [58] Arvind et al. have demonstrated a way to measure CPU temperature using on die thermal sensors. The readings of the temperature sensors of the test quad core CPU are stored in MSR: `IA32_TEMPERATURE_TARGET` and accessible with `lm-sensor` from Linux. The on die sensor oriented software temperature monitoring method is shown to be able to monitor the temperature variation with a 10ms interval with negligible amount of overhead. The power dissipation throughout a CPU core is not uniform at runtime as different CPU components such as cache, buffer, ALUs and FLUs generates different amount of power dissipation as the density of logic gates used to construct them are different. In [59] Ardestani et al. have addressed this problem by proposing the power density and thermal aware floorplanning which is aimed to eliminate as much predictable thermal and power dissipation problem as possible at the design stage. The mean time to failure models are utilized to gauge the thermal tolerance of different floorplan layouts. The result has shown that by considering the power density of individual CPU components, it has reduced the

leakage power by 3% to 7% which is indeed a trivial amount of power reduced if the additional overhead is considered to implement similar power computation method at runtime.

Dynamic Thermal Management techniques have been developed to mitigate the hotspot problem in CPUs. In [60] Salami et al. have presented a thermal aware task migration algorithm in a Simultaneous Multi-Threading (SMT) multi-core processor. The algorithm actively engages different CPU cores to determine the possible transient temperature imparities that vary across different CPU cores. To compute the amount of thermal dissipation generated by a given CPU activity the algorithm monitors a series of performance counter readings monitored during the CPU activity. Each performance counter is assigned with a preset coefficient to deduce the temperature of the CPU core after handling the activity. However the amount of temperature reduced by the algorithm is a marginal 1.1 to 1.3% comparing to other temperature managers. Thus the scheduling approach proposed in the thesis considers both power dissipation and thermal dissipation as scheduling decision constrains to achieve better performance gain and power dissipation reduction.

2.5 Power Management in Heterogeneous Architectures

Heterogeneous processor architecture has become popular over the years as an effective mean to resolve power wall problems that cannot be easily managed by homogeneous architectures. To adapt the design of the Scheduling approach proposed in this thesis towards heterogeneous architectures, the following research works have been studied. In [61] Lukefahr et al. introduce the composite core which turns a single core into a heterogeneous structure with big and small processing engines. Both engines shares the same instruction fetching unit, branch predictor and data cache with two separated decoders. The focus of the design is to mitigate the switching overhead in a fully heterogeneous system that comprehends asymmetrical clusters of identical CPU cores. However the result shows that overall performance has been compromised to achieve the savings in energy. In [62] Lukefahr et al. have developed an offline analysis tool to determine optimized DVFS rules for a given heterogeneous processor. Performance requirements of different applications are learnt offline, the amount of energy dissipated and the amount of CPU time utilized by each application executed in a different inter core switching pattern are recorded to compute their individual quanta. The quanta are then stored in a tree and used to make scheduling decisions based on the computed minimum quanta. The biggest drawback of the approach is that it does not support online adaptation of the quanta computation and it is unlikely that an offline based approach is able to cover as many as scheduling problem in practice. In [63] Kahng et al. demonstrate a context aware multimodal DVFS design approach. In the study the relationship of DVFS and power dissipation overhead generated has been discussed. The amount of overhead generation is positively correlated to the difference between the peak operating frequency and the lowest operating frequency. The overhead increases as the amount of CPU time operating in the peak frequency increases. Hence the OPP utilizable in the DVFS is grouped into different modes according to the frequency and power dissipation overhead, and the modes are used to conduct context aware DVFS. The result shows a 10% to 5% of power reduction, however this approach also requires the DVFS modes to be computed in an offline process. In [64] Francisco et al. propose a framework of application aware task management for heterogeneous platforms. The framework provides the Linux kernel with a historical system resource usage tracking mechanism based on

active user space application reporting. With the help of the application tracking information various power management technique will become task aware. However the approach shows promising result only when all applications in user space do note the significance and make use of the API. In [65] Nollet et al. show a novel low cost task migration technique utilizing debug registers in the CPU. The technique decreases the task migration reaction time by storing migration handler so that the task can be executed in absence of migration request checks and carries out migration once it is initiated by the handler with help from the Debug register.

Scheduling in heterogeneous architecture has demonstrated more commercial potential rather than a mere research interest. There are a handful of heterogeneous system targeting scheduling approaches have been deployed in the mobile computing products. Linaro had designed the Global Task Scheduling (GTS) [66] to accommodate the possible performance boost of ARM's big.LITTLE [67] architecture. In which all of the cores in the asymmetrical processor are able to process tasks in parallel, tasks can be allocated freely to individual CPU core in both clusters. However there are many limitations in the actual execution of the GTS scheduler: 1. the boosting mechanism is matched with existing `cpufreq` governors where only limited number of voltage islands are available, excessive power dissipates through the computation process as finer grained task to voltage island mapping is not available. 2. The triggering of scheduling scheme relies on application requests raised solely from the user space, as the workload capacities were calculated based on events, which resulted in possible problems such as, upon touch event all application threads will be prioritized to be allocated onto the big core, regardless of computation power required by the foreground application. Due to limitations of the GTS, the main stream of big.LITTLE SoCs are now equipped with the In-Kernel Switcher (IKS) [68]. Each little core is grouped with a big core to form a virtual CPU pair. Only one core will be activated at one time and application thread metrics are monitored for the scheduler to make thread migration decisions between two cores in a virtual pair. Given the fact that mobile computing platforms normally execute less computation intensive applications that require the entire processor to run at its full capacity, the IKS has demonstrated its ability in managing the power dissipation effectively. However, with only half of the cores activated at any given instance, it is indeed a wasteful approach to utilize the computing power of a system. Hence in this thesis, it had been researched into the idea to accommodate more clusters in a SoC to suit different needs of applications and fewer cores in each cluster so that all cores can be affordably powered up together. The Energy Aware Scheduling (EAS) [69] was developed natively to boost the computing power with asymmetrical processor in mind which is a remarkable effort to seamlessly combine the DVFS functionality with the software intervention together. The project aimed to combine `cpufreq` and `cpuidle` together to achieve a universal power to performance model. Progressively the development of EAS is working towards making the DVFS scheduler driven, which is in the same direction of the scheduler proposed in this thesis.

There are many research works aimed to apply machine learning in Linux scheduler. In [70] machine learning has been applied to calculate the optimized time slice length in Completely Fair Scheduler (CFS) scheduler [150] and had achieved 5% of overall improvement in the turnaround time. In [71] a multilayer perceptron (MLP) scheduler has been developed with the aid of M5 learning tool, and had achieved up to 7% of improvement compared to the state-of-the-art schedulers. However the existing literatures have not tailored machine learning functions to suit their schedulers and require off-line learnings for initialization. In contrast the Scheduler proposed in this thesis developed dedicated machine learning function blocks to minimize computation overheads and kept all learning process online to maximize its compatibility.

2.6 Neural Network Applications and Optimization

The applications of deep neural networks (DNNs) have gained growing interests with the universal approximating ability in solving complex problems with non-parametric solutions. In [72] Lim et al. develop a system of multiple neural networks to solved pattern classification problem with an online approach. In the system a number of different neural networks are used as building blocks, each of which functions as a classifier. The output of each classifier is combined with behavior knowledge space algorithm. The system is applied with both online and offline learning approach, and the result has been compared between learning using training set in original format and features extracted. The result shows an enhancement in prediction accuracy by combining the result form different classifiers, and the computation efficiency of the online learning is affected as classification outcomes from multiple classifiers are computed.

Deep Neural Networks DNNs can be constructed with different neural network components as basic building blocks. In [73] Zhang et al. proposed a (DNNs) to solve low resource recognition problems. The DNN is constructed as a two-stage network and each stage is a hidden Markov model consisting of 8 layers with 6 hidden layers. The sixth layer is a bottleneck layer that reduces the dimensionality of the features. The last layer is a softmax layer to generate classification output. The two-stage network design managed to divide the problem into high rank and low rank, which is assumed to be more efficient due to the sparse distance between the two ranks in the data space. The DNNs are pre-trained with restricted Boltzmann machines (RBMs) and fine-tuned with backpropagation. The only drawback of the approach is that the performance gain is not proportion to the amount of additional hidden nodes created due in the two-stage network. In [74] Fischer et al. have reintroduced the RBM by emphasizing on the different training approaches and precautions. RBMs have attracted attention as building blocks for deep belief networks (DBNs). The learning of RBMs are based on the Markov chain Monte Carlo methods and contrastive divergence is one of the most popular learning algorithm to date. The RBMs are representation of probabilistic graphical models that describe probability distributions by mapping conditional dependency and independence properties between random variables using graph structure. Where the conditional distribution of the given sets of random variables get shifted towards each other with the process of Gibbs sampling. Compare to Autoencoders, the RBMs have better generalization efficiency in terms of determining results for indiscriminative problems. In [75] Hinton discusses the learning multilayer generative models as an alternative DNNs learning approach to the vanishing gradient restricted backpropagation algorithm. RBMs has been adopted in the process of learning multiple layers at once, and has shown promising result against deep MLP with backpropagation learning as especially in the process of modeling sequential data where conditional distributions from more random data sets are being accessed. In [76] Ngiam et al. have explored the learning capacity of unsupervised DNNs over training set from multiple modalities. The study has demonstrated cross modality feature learning that benefits the features learnt with a better accuracy by considering the presence of correlated features from another modality. The dynamic of the theory is rather straightforward and similar compare to the multi classifier network and the additional feature pegged to the observation unit in DMNN, which includes information that provides additional dimension to the

training set do increase the efficiency of the learning process. RBMs and Autoencoders have been utilized as the basic building blocks of the DNN. Features are extracted with individual RBM or Autoencoders and the output is fed to a shared layer to generate the correlated features of training set from different modalities. The result has demonstrated the multimodal approach does not always outperform the single modal learning methods. Autoencoders in some cases do perform differently and outperform the RBMs. In [77] Chandra et al. explore the usage of Autoencoders in conducting unsupervised feature selection. In the learning of a large training set with a very high feature to pattern ratio, good feature selection is rather one essential factor to increase the efficiency and accuracy of the network. To select the features in unsupervised manner with the help of Autoencoders, it is able to reduce the amount of features that is less prominent to the given label. The result shows that Autoencoder based feature selection boosts the performance of the supervised MLP by 10%. In [78] Makhzani et al. have introduced a sparse Autoencoder to improve classification results by encouraging sparsity. In their Autoencoder proposed, only a number of highest values are able to activate the hidden neurons and trigger the update of the connecting weights, which indeed functions as a constraint term that controls the activation of neurons to avoid over fitting and adds sparsity between classes. In [79] Marivate et al. discuss the Autoencoder and Principal Component Analysis for data imputation. Data imputation refers to the method used to deal with missing value and variables in the data collected. The study has shown that with the best performing data imputation method is the (principal component analysis) PCA aided Autoencoder, where the data is at first processed with the PCA before feeding into the neural net. In [80] Amaral et al. have compared the performance difference of using different cost functions during the pre-training and fine-tuning phase of stacked Autoencoders. The performance of cross entropy (CE) and sum of squared errors (SSE) cost functions have been compared. The result shows the SSE performs the best during the layer-wise pre-training and the CE is better at fine-tuning. In [81] Larochelle et al. propose a new approach to model the distribution of high dimensional vectors of discrete variable, which is indeed a process of using Autoencoders to deduce conditional distributions in a RBMs like manner. The advantage of the method is that, the joint distribution becomes visible and tractable and maintaining the capability of assigning probability to observations.

Different combinations of DNN building blocks can be utilized to solve problems dynamically. In [82] Deng et al. have addressed the deep stacking network (DSN) that exploits the parallelism in many-core multi-core computing system. The DSN is realized with the stacking concept where simple classifier blocks are stacked to learn complex functions. The output of the upper block combines with the input of the lower block as the input of classifier in the lower block. The advantage of DSN is that it is able to compute large data set without reaching out for GPU due to its highly paralleled parameter computation method. In [83] Lendaris has proposed a novel approach to employ adaptive dynamic programming with the help of an online learning algorithm. The approach is named as the higher level learning algorithm that instead of repetitively improves the system performance through reinforced learning now conducts a higher controlled optimization of the system with supervised learning methods such as the Multi-Layer Perceptron (MLP). The approach has shown its efficiency in improving the system performance in a higher perspective. However its vastness prevents its adoptability in localized dynamic programming problems. In [84] Baldi et al. have discussed the application of multilayer neural network in mapping the function of PCA. The neural network is learnt with backpropagation method, and a quadratic error function has been utilized to update the value of the connecting weights. At the end of the learning phase the landscape of the error function is visualized and reconstructs different gradient

decent steps taken to look for the global minima. The result has demonstrated the assumption that the local minima and global minima of the error function corresponds to the orthogonal projection on the subspace spanned by the principal eigenvectors of one of the covariance matrix that is associated with the training set. The study has bridged the early connection between PCA and neural network. In [85] Wei et al. have addressed the issue in solving search-based problems with a vast search space due to states that grow exponentially. Such kind of state explosion can easily and rapidly occupy the memory of the processing platform and cause additional computation time. Hence a state validation approach has been developed in the study, where the evaluation values of a state in the current optimized path will be reduced. As a result the states that are likely to provide promising results get continuously moved forward. In [86] Yang et al. present an approach to classify sequential data using Deep Markov Neural Network (DMNN). The classifier is constructed with two parts, the hidden Markov component and the recursive neural network component. In the DMNN model each observation sampled across a continuous series of time is given an additional feature, and based on the given feature and observation a targeting label is supposed to be predicted. The accuracy of the prediction is merely 70%, which is rather expectable as the training set used in the learning phase is rather small.

One of the important factors to consider for neural network optimization is the selection of features prior to the training phase. In [87] Leray et al. address the method to select feature based on given task. Neural network specified selection methods have categorized into three different groups according to their relationship with the network parameters. The first order saliency based pruning method has shown good synergy with the neural network optimization approach proposed in this thesis. In [88] Yang et al. present a feature selection method for MLP using feature ranking approach. The features are ranked by computing the difference of the output value of a class before and after a feature is removed from the training set. The larger the difference the more significant the feature is. Instead of retrain the MLP to testify the significance of each feature, random permutation approximation method is adopted to avoid expensive retraining with a certain compromise in the accuracy. In [89] Dorffner discusses the processing of time series with neural networks and forecasting based on the learnt result. Linear trend removing has been first applied for data preprocessing. The result has shown that the recurrent neural network provides the best forecasting result amongst all the candidate networks. However the prediction window has to be carefully chosen to avoid over complicated weight computation at training phase. In [90] Mörchen presents a dimensionality reduction method for time series data, where each time series is decomposed with Fourier transform and the most prominent set of coefficients is selected. The algorithm used to determine the prominent features is the k means clustering. However Autoencoder aided PCA is able to select the features with similar accuracy and higher efficiency in a fully unsupervised manner.

The pattern of incoming data spanned through time space is one of the particular interest to the proposed methods in this thesis, and DNNs can be utilized to predict time series sequential data with the help of recurrently constructed neural networks. In [91] and [92] Martens et al. have resolved the problem of training recurrent neural networks (RNNs) that contain long-term data dependencies efficiently. The Hessian free (HF) optimization approach has been adopted in the study, which is an operation of continuous iterative minimization of a simpler sub-objective based on the local approximation of the given objective. Thus the RNN is able to be deployed as better time series forecasting method with better efficiency. In [93] Pascaru et al. have shown the procedure and precautions of setting up a deep RNN. Three setups has been proposed by their study, the deep transition RNN where hidden neuron value is approximated with MLP instead; the deep output RNN where the

output neuron value is approximated with MLP; and the stacked RNN that multiple RNNs are stacked where the hidden state value is computed with the input neuron value instead of hidden neuron value from previous instance. In [94] Shiraga et al. have proposed an online incremental learning reinforcement learning (RL) approach, where a long short term memory network (LSTMN) has been deployed, to reconstruct the execution steps of a RL agent. Being incremental the LSTMN is to face interference as comparing to mini batch based learning. However interference can be mitigated by the memorized data in the network. In [95] Gangireddy et al. have proposed a MLP based RNN pre training approach. The MLP functions as feed forward neural network with a softmax layer at the output. The hidden layer of the MLP and the RNN are both connected to the softmax layer of the MLP during pre-training. As a result the hidden layer of the RNN will be pre trained with backpropagation algorithm.

Chapter 3 User Pattern Aware Scheduling Schemes

3.1 Introduction

The motivation behind the current power dissipation aware scheduling schemes discussed in Section 2.2 is the belief that the system performance fluctuates randomly during the process of system resource allocation. The influence of different applications on the power dissipation patterns is neglected. The proposed work of constructing application usage driven user pattern aware scheduling schemes discussed in this chapter is based on the hypothesis that different users possess different but consistent habits of accessing the same set of applications. Such patterns are recognizable by the system and can serve as a basis to predict the workload at the next instance. The relationship between applications usage pattern and system power dissipation had been overlooked in the discussions of the existing power management literatures.

Hence in this chapter, a group of scheduling schemes is proposed with the primitive focus on utilizing user power profile recognition to increase the accuracy of the applications workload prediction. In the proposed power dissipation aware scheduling schemes, different applications are monitored and analyzed as a group at a given process interval. Their influences on the system IO, the system memory and other system resources are converted to power relationships through model simplification. The model describing the relationship is named as the Thread Interference Power and Thread Solitary Power (TIP-TSP) model. The TSP indicates the amount of power dissipated solely by executing individual applications and TIP indicates the additional power dissipated by applications executing in parallel. Dedicated power management schemes that conserve the most power dissipation are chosen based on the power profile established through the TIP-TSP model. Such schemes and application usage pattern pairs are kept in the profiling tables for future predictions.

Similar progress is repeated at the process interval for the same or a new set of incoming applications. The accuracy of the prediction will be improved as the profiling table builds up. As a result the number of predictions and runtime analysis decrease over time. The overall power consumption is saved not only by the fine-tuned management schemes but also the less overhead produced in the runtime application pattern analysis. The GEM5 [96] simulator was used for system performance simulation and McPAT [97] was used for power simulation in this work. It was observed from the results of the simulation performed that the results obtained matched the proposed hypothesis of reducing power dissipation with application profile and management schemes pairs.

The power management schemes proposed by various studies to solve the dark silicon problem are mainly based on homogeneous architected multi-core system. The directions of their studies based on predictions of system events like scheduling phasing, power phasing and cache missing rates without generalizing the events with higher level analysis. Most of the system events are instantaneous and has shown difficulty in establishing a long term accurate prediction basis. This problem is caused by the ease of adoption of general purposed system designing approaches. In the approaches the system events are not planned with consideration of individual applications, all system resource allocation decisions are made by the system scheduler. In the heterogeneous designs the possibility of increasing performance and reducing power dissipation of dedicated operations with specialized cores are demonstrated, which had inspired the proposed scheduling schemes in this chapter to manage each combination of applications differently based on their distinctive behavior. The proposed

scheduling schemes are built based on the belief that the applications usage patterns monitored at higher level can directly reflect system power dissipation.

To conclude, the proposed scheduling schemes are designed to manage the system power dissipation with improved prediction learning accuracy and system resource allocation efficiency.

The main tasks described in this chapter are as follows:

- Establish the Thread Interference Power and Thread Solitary Power (TIP-TSP) model to monitor the user usage pattern and power dissipation relationship.
- Construct user power profiling lookup table named *AppSet Table* for prediction and optimization purposes.
- Create adaptive scheduling schemes that allocate system resources according to TIP-TSP model and the *AppSet Table*.
- Test the feasibility of the TIP-TSP model and the proposed scheduling schemes in comparison to the existing power management schemes.

The rest of the chapter is organized as follows. Section 3.2 presents the structure of the power management approach. Section 3.3 presents the detail of the power management schemes used in the power management approach. Section 3.4 presents the simulation setup and simulation results. Section 3.5 concludes the chapter with future work plans.

3.2 Proposed Power Management Approach

This section presents the proposed approach for the user pattern aware scheduling schemes. The schemes of this study are realized by implementing scripts running in parallel with the system thread scheduler in the Operating System (OS) Linux. Figure 3.1 shows the overview of the approach for the proposed schemes, where the *Monitor* script records power related parameters while the threads are in the scheduler queue indicated with dotted lines as shown in Figure 3.2, where T_m , T_{m-1} and T_{m-2} are the thread scheduling intervals for the threads in the scheduler queue. The *Updater* script is called by the *Monitor* script to update the parameters collected and recorded in the corresponding tables. The parameters recorded are used by the *Optimizer* script to compute the optimized power management scheduling sequence. The *Scheduler* script decides the actual thread scheduling sequence based on the optimized scheduling sequence when the scheduling request reaches the scheduler as the solid line indicates in Figure 3.1. The *Interleaver* script is occasionally called by the *Monitor* script to create a special scheduling interval with optimized sequence computed by the *Optimizer*. The detailed power component monitoring procedures, functions of each script and examples of deducing power components are discussed in Section 3.2. The terms ‘application’ and ‘app’ are used interchangeably in the remaining chapters of the thesis.

Figure 3.1 Overview of the management schemes

 Multithread Scheduled Application Time Slices at interval T_m

Figure 3.2 Multithreaded application threads

3.2.1 User Power Profile Recognition

In this work, the user power profile refers to the power dissipation pattern for a user using apps on a mobile device with multi-core processor. The user power profile recognition is the key component of this work. The concept is based on the fact that only less than 10 apps are used by average mobile device users every day [98]. The app usage habits of different users could be different and repetitive. For example, a business man may spend his weekday mornings with Email app to arrange meetings and Office app to access business documents. On the other hand, a student user's morning apps could be the web browser for the latest news and the media player for music on the trip to school. These habits assemble an app usage pattern for each user and this work studies on recognizing the usage pattern to build a power profile for each individual user. These profiles would well reflect the total energy consumed by users at different times of a particular day. Predictions about the

applications running in the next moments can then be made based on these profiles for application aware power management. By analyzing the user power profiles, a method to minimize the power consumptions with thread scheduler at appropriate intervals can be derived. The user power profile recognition studied in this work is on an ARM platform running the Android operating system. As shown in Figure 3.2, the scheduler in Android divides a scheduling interval into different frames named time slices. The time slices are allocated to different threads from different applications. The applications can be given priorities by the system or set by the programmer [99]. With higher priorities an application will be allocated with more time slices. For example the background system threads are given the highest priority, and a moderate priority is given to all third party user interactive applications with low IO demands. This operation will realize runtime multithreading. In order to minimize the impact of the proposed scheduling schemes towards the native Android scheduler, the study focuses mainly on the user interactive applications.

3.2.2 Thread Interference Power and Thread Solitary Power Model

In order to establish the user power profile, the application related power components are first identified. The power dissipation caused by the running applications in an operating system consists of two parts, the Thread Solitary Power (TSP) and the Thread Interference Power (TIP) as modeled in the TSP-TIP model formulated by Equation (3.1).

$$P_s(n_k) + P_d(n_k) = P_t(n_k) \quad (3.1)$$

where

P_s : the Thread Solitary Power (TSP),

P_d : the Thread Interference Power (TIP),

P_t : the total power dissipated, and

n_k : k times of threads dispatched by the n^{th} application.

The TSP contributes a larger portion of the overall power consumption in a system [100]. The TSP occurs in several events:

- The arithmetic operation, the control and logical activities occurred in the CPU cores.
- The read and write operations that take place in the caches.
- Handling the data storage requests in DRAM.

The TSP can be modeled with Equation (3.2).

$$P_s(n_k) = P_{cache}(n_k) + P_{mem}(n_k) + P_{pipe}(n_k) \quad (3.2)$$

where

P_{cache} : the power dissipated in the caches,

P_{mem} : the power dissipated in the memory, and

P_{pipe} : the power dissipated in other parts of the CPU pipeline.

TSP dissipated by the threads of the same applications follows the same pattern, which is due to the fact that the state of art multi-core multithread systems exploit the parallelism by distributing the task into equivalent threads. For example, to compute the sum of four values $x5=x1+x2+x3+x4$, the main thread can choose to split

the process into three equally loaded threads: $y1=x1+x2$, $y2=x3+x4$ and $x5=y1+y2$, and then schedule the operations into different CPU cores. Parallelism will only speed up the process the most when the threads are evenly loaded. Thus the operation loads are similar amongst threads from the same application. The threads will make use of the same CPU pipeline to accomplish the task and acquire similar amount of memory, hence dissipate TSP in the similar pattern. The overall TSP consumption of multiple threads under the same application is an aggregation of the consumption of a single thread as in Equation (3.3) and cannot be easily reduced.

$$P_s(n_k) = P_s(n) \times k \quad (3.3)$$

where

n : the number of the applications monitored, in sequential order, and

k : the number of running threads of the n^{th} application.

The TIP occurs during the following stages:

- The scheduler has threads of multiple applications in queue that the currently scheduled threads have to be rescheduled to new time slices.
- The repetitive read and write operations of caches while the multithreading is performed. The caches will be emptied and header files marking the thread running status with the thread will be dumped.
- The allocation of memory to store the thread data and the headers [101].

The TIP can be modeled with equation (3.4).

$$P_d(a, b) = [P_{refresh}(a, b) + P_{header}(a, b) + P_{data}(a, b)] \times f_t \quad (3.4)$$

where

$P_d(a, b)$: the TIP dissipated by multi scheduling threads from the applications a and b at each time slice,

$P_{refresh}(a, b)$: the power dissipated in refreshing the caches for thread belong to the applications a and b ,

$P_{header}(a, b)$: the TIP dissipated due to header created for the applications a and b ,

$P_{data}(a, b)$: the TIP dissipated in memory allocation storing runtime data of threads from the applications a and b , and

f_t : the multithreading frequency, number of times slices a thread will be allocated in a scheduling interval.

The TIP components ($P_{refresh}$, P_{header} , and P_{data}) of the application i are time slice sequence dependent. The TIP dissipation occurs at the moment of the threads switching. The amount of power dissipated is determined based on the value of pipelining stages, thread size and IO demand of two threads in neighboring time slices. As a result, different applications in the neighboring time slices will cause different TIP dissipation. As shown in Figure 3.3, the TIP dissipated over the interval is different since the sequence is different. To compute TIP for all combinations of time slices will increase the computation complexity greatly.

Figure 3.3 Time slice sequence 1 and 2

To reduce the computation complexity of the power dissipation and for the feasibility of this study, the relationship between the Thread Interference Power and Thread Solitary Power are simplified as (3.5). The simplified model computes the TIP between different application combinations instead of between each time slice. As such, the parameters used to monitor power dissipation are reduced and yet capable of providing accurate computation with minimum uncertainty.

$$P_d(a, b) = P_t(a, b) - P_s(a, b) - \varepsilon_d(t) \quad (3.5)$$

where

$P_d(a, b)$: the TIP between application a and application b ,

$P_s(a, b)$: the total TSP of application a and application b ,

$P_t(a, b)$: the power of application a and application b , and

$\varepsilon_d(t)$: the fluctuation coefficient of the TIP components, deduced based on the user application profiling.

This study focuses mainly on the exploitation of TIP dissipation among different application combinations. In the following sections, the implementation of user profile monitoring is detailed.

3.2.3 Monitor the Scheduling Activities

The Thread Interference Power (TIP) and Thread Solitary Power (TSP) components of each application combination are monitored at every scheduling interval. The monitor function is implemented in the scheduler schemes as the *Monitor* script and tested in simulated environment. Different combinations of multithreading applications at every scheduling interval are collected and named as *AppSets*. The *Monitor* script collects the runtime *AppSets* data, such as, the total TIP, the total TSP and the occurring frequency of *AppSets* upon creation and expiration of each thread, and stores the data in the *AppSet Table*. The script *Updater* updates the TIP components, TSP components, thread sizes, interval length and occurring frequency of the *AppSets*. The detail about the *Updater* is elaborated in Section 3.1.3. The *Monitor* script is explained below as Script 1.

SCRIPT 1 MONITOR

```

1: Upon thread scheduling, events are true. Set  $e = 1$  at instance  $T_m$ 
2: WHILE  $e = 1$ 
3:   CALL Updater,  $Proc(T_m)$ 
4:   IF first application
5:     CALL Updater,  $P_d(n)=NA., P_s(n)=P_t(T_m)=P_b., F(n)=1$ 
6:   ELSE
7:     IF  $n$  exists in the AppSet Table &&  $Count(Proc(T_m))=1$  THEN
8:       CALL Updater,  $P_t(T_m), F_s(n)++$ 
9:     ELSE
10:      IF  $Count(Proc(T_m))=1$ , THEN
11:        CALL  $P_d=P_t(n)-P_s(n), TSize, F_s(n)=1$ 
12:      ELSE
13:        FOR not the last application in AppSets,  $Count(Id(T_m))=++$ 
14:          CALL Updater,  $P_s(n)=P_s(n)+P_s(n+1), TSize, F_s(n)=1, i++$ 
15:          CALL Updater,  $Proc(T_m)$ 
16:        END FOR
17:        CALL Updater,  $P_d(n-i, ..., n)=P_t(T_m)-P_s(n), F_s(n-i, ..., n)$ 
18:      END IF
19:    END IF
20:  END IF

```

21: $e=0$
 22: CALL *Updater*, $L(T_m)$

where

$P_d(n)=NA.$: the script dose not calculate TIP for the first application, hence the value is not applicable.

$Proc(T_m)$: the application ID present at the interval T_m . It is used for *AppSet Table* to relate the corresponding parameters stored.

$Count(Proc(T_m))$: the number of applications in the current monitored *AppSet*.

$F_s(n)$: the number of times that threads from the same application had been scheduled.

$TSize$: the average size of a thread of the same application.

T_m : the interval time stamp.

$L(T_m)$: length of interval T_m

e : living thread indicator, $e=0$ once thread exits

P_b : background power indicates the base value of the current *AppSet Table*.

$P_t(T_m)$: in the case of variation of P_t values of the same *AppSets*, the *Updater* would update the range of the values.

As shown in Script 1, upon activation of the *Monitor* script the system power consumption without running user interactive applications is the background power P_b . Any threads scheduled before recording the value of P_b will be ignored by the monitoring script and will not be updated in *AppSet Table*.

AppSet Table is updated sequentially. Each newly monitored *AppSet* will be issued with a sequence number n . The sequence number simplifies the search and comparison functions in the *AppSet Table*. *AppSet* values can be retrieved from the table with their sequence numbers n , or the combination of application *ids*. For *AppSets* with unknown TSP, an equation representing the relationship between TIP and TSP components instead of numerical values will be passed to the *Updater* to update in *AppSet Table*. Equation (3.6) is an example of such equation describing the power component relationship for *AppSet(a,b)*,

$$P_d(a,b) = P_t(a,b) - P_s(a,b) \quad (3.6)$$

3.2.4 Updating *AppSet Table*

Once the *Updater* Script is called by the *Monitor* Script, it will response according to the application. As shown in Script 2, the *Updater* will add new entry to *AppSet Table* if the new *AppSet* is not previously seen. If the incoming scheduling event carries similar *AppSets* with matching records in *AppSet Table*, the *Updater* will check if there is any difference in the total power. In the case that the new total power is different from the old one, the *Updater* will perform an arithmetic operation to average the values. The *AppSet Table* also contains equations instead of values if the TSP of *AppSet* is unknown. TSP values are required to compute TIP, in some cases the required TSP values are not known at the point of TIP computation and can only be obtained when it is monitored at a later scheduling interval. Hence the TIP values are updated as equation instead of

values and once the relevant values are acquired by the *Monitor* script, the *Updater* Script will update the value in *AppSet Table* globally.

SCRIPT 2 UPDATER

```

1: Upon CALL Updater at scheduling events, set  $e=1$  at instance  $T_m$ 
2: WHILE  $e = 1$ 
3:     IF  $Proc(T_m) = false \ \&\& \ n=0$ 
4:         RETURN 1st application,  $n(Proc(T_m))=1$            # the AppSet is empty, issue number
1      1
5:     ELSE
6:         IF  $id(T_m) = false$  THEN                             # first thread of this application monitored
7:             RETURN  $n(Proc(T_m))=n+1$                        # issue sequence number for  $Proc(T_m)$ 
8:         ELSE
9:             IF  $n=1$  THEN
10:                WRITE TO AppSet( $n$ ),  $P_d(n)=NA.$ ,  $P_s(n)=P_t(T_m)=P_b$ ,  $TSize$ 
                                     # add new line into AppSet Table.
11:            ELSE
12:                IF  $P_t(T_m) \neq P_t(n)$ 
13:                    WRITE TO AppSet( $n$ ),  $P_t(n)=P_t(T_m)$ ,  $P_s(n)$ ,  $P_d(n)$ 
14:                ELSE
15:                    WRITE TO AppSet( $n$ ),  $P_t(n)$ ,  $P_s(n)$ ,  $P_d(n)$ 
16:                END IF
17:                FOR  $i=1$  to  $i=n$ 
18:                    IF  $P_s(n)$  is needed in AppSet( $i$ )
19:                        UPDATE AppSet( $i$ ),  $P_s(n)$ 
20:                    END IF
21:                END FOR
22:            END IF
23:        END IF
24:    END IF
25:    WRITE TO AppSet  $L(T_m)$ 

```

where

$id(T_m)$: the *AppSet* id of the process monitored at interval T_m .

$n(Proc(T_m))$: the *AppSet* table sequential number for $id(T_m)$, which is issued base on monitoring sequence,

AppSet(i): the *AppSet* values in row i ,

WRITE: write and create new entries in *AppSet Table* cells

UPDATE: replace the update existing entries in the corresponding *AppSet Table* cells with new values.

While updating *AppSet Table*, the TSP of an application is calculated in an aggregation manner. Upon logging of a new *AppSet*, the thread size of the first thread of the application will be updated with the *Updater* script as the Unit Thread Size. While the subsequent threads scheduled for the same application would have their TSP calculated based on the fraction of their new thread size and the Unit Thread Size $S_t(n_1)$ as in Equation (3.7).

$$P_s(n_2) + P_s(n_1) = P_s(n_1) \times \frac{S_t(n_2)}{S_t(n_1)} \quad (3.7)$$

3.2.5 Quiet Intervals

The Thread Solitary Power (TSP) of an *AppSet* can be calculated only when negligible Thread Interference Power (TIP) is present. The scheduling intervals with one thread running in a CPU core are ideal moments for the TSP calculation. Such intervals are named as the quiet intervals because the TIP dissipation caused by the multithreading is absent. Quiet intervals can occur intrinsically or be created with the *Interleaver* Script. As shown in Script 3, the *Interleaver* script makes use of scheduler functions to hold the dispatching of certain threads of one or more applications for a brief duration determined as explained below.

Creating quiet intervals will reduce the TIP dissipation but increase the overall TSP dissipation due to the delays in scheduling. To optimize the length of the delay, the *Interleave* script determines when to create quiet intervals and their durations based on the information provided by the *Optimizer* script. The *Optimizer* script determines the thread scheduling based on the TSP and TIP values logged in *AppSet Table* and returns to the *Interleaver* an optimized length of scheduling the quiet interval. The optimized length computation is based on the TIP dissipation reduced and the TSP dissipation increased. The details of the *Optimizer* script are presented in Section 3.3.2. The *Interleaver* script will stop creating quiet interval once the intended TSP is monitored or the optimized length is expired.

SCRIPT 3 INTERLEAVE

```

1: Upon quiet interval counter expiration  $c(n)=0$  at instance  $T_m$ , this timer indicates numbers of intervals that
   the script should wait before making quiet intervals for AppSet  $n$ .
2: WHILE  $c(n) = 0 \ \&\& \ Proc(T_m)$  contains AppSet( $n$ )
3:     FOR  $d < 1, d=5, d--$ 
4:         COMPARE  $Proc(T_m), AppSet$       # to check if all queuing applications are known
AppSets
5:         IF all applications are known AppSets
6:             CALL Optimizer, AppSet(Proc(Tm))      # ask Optimizer script to calculate
intervals and sequences that AppSet( $n$ ) can be scheduled
7:             SCHEDULE AppSet( $n$ ),                # Schedule the AppSet( $n$ ) and halt other
threads (or use cpu affiliation, to contain the quiet interval in one CPU core.)
8:             WAIT  $L(o)$  or  $P_s$  monitored          # Wait for an optimized period of time
9:             SCHEDULE Normally                    # Resume the scheduling of other AppSets.
10:            CALL Monitor, Proc(Tm)              # Call Monitor script to learn the unknown
AppSet
11:        END FOR
12:        CALL Optimizer, AppSet(Proc(Tm))      # After 5 (can be modified for optimized
performance) attempts, force schedule AppSet( $n$ ) bases on current AppSets table
13:        SCHEDULE AppSet( $n$ ),
14:        WAIT  $L(o)$  or  $P_s$  monitored
15:        SCHEDULE Normally
16: RETURN  $c(n)=default$                         # Set  $c(n)$  with default value

```

where

$c(n)$: the quiet interval counter, issued to each unknown TSP component upon logged in *AppSet table*,

d : indicates the numbers of tries before the forced scheduling is applied,

COMPARE: searches the application *ID* in the *AppSets* for each application in the scheduler,

SCHEDULE: dispatches a single thread from *AppSet*(n) to the scheduler, and

WAIT: wait function, creates an interval between the current operation and the next.

3.2.6 An Illustrative Scenario

An illustrative example for the quiet intervals, the computation of power and the updating of *AppSet Table* is given here. Figure 3.4 shows a scenario with two quiet intervals $t1$ and $t3$, in which only one thread is scheduled. During the quiet interval, the *Monitor* script will log *AppSet Table* with power dissipation equations. Once the quiet interval is reached, the *Monitor* script will acquire the Thread Solitary Power (TSP) and complete the computation of the Thread Interference Power (TIP).

Figure 3.4 An illustrative scenario with quiet intervals

In the case where the quiet interval rate drops due to large number of scheduling requests, the *Monitor* script will call for the *Interleave* script to create a quiet gap similar to $t3$, by either holding back the scheduling of queuing threads or suspending the running threads. The *Interleave* script will decide when to create quiet intervals by considering the impact that may cause the performance to degrade. Once all the TSP values in *AppSet* are acquired, *AppSet Table* will be established as Table 3.1. The *Updater* script will continue to update *AppSet Table* at the beginning of each interval. New *AppSets* will be added to the table. The existing *AppSets* will be updated if the total power monitored in the current interval is different from the recorded value. For example, the *AppSet*($a1, b1$) is updated dynamically with the running average power since there are two occurrence of *AppSet*($a1, b1$) (the intervals $t2$ and $t4$ in Figure 3.4) and $P_t(t2) \neq P_t(t4)$. As a result, the $P_d(a1, b1)$ is recalculated with equation (3.8).

$$P_d(a1, b1) = \frac{P_t(t2) \times O(t2) + P_t(t4) \times O(t4)}{O(t2) + O(t4)} - P_s(a1, b1) \quad (3.8)$$

where

$O(t2)$: the number of times of $P_t(t2)$ occurred in *AppSet*($a1, b1$), and

$O(t4)$: the number of times of $P_t(t4)$ occurred in *AppSet*($a1, b1$).

Table 3.1 *AppSet Table* for quiet interval scenario

AppSets	TIP (P_d) (Watt)	TSP (P_s) (Watt)	P_t (Watt)	Interval length (L) (ms)	Thread Size (T_{size}) (KB)	Range	Frequency
a1	NA	0.3	0.3	500	600	NA	5
a1,b1	0.3	1.0	1.3	400	NA	0.05	3
b1	NA	0.7	0.7	100	700	NA	4
a1,c1	0.5	0.7	1.2	100	NA	0.05	2
a1,b1,c1	0.8	1.4	2.2	100	NA	0.1	1
c1	NA	0.4	0.4	100	400	NA	3

The *AppSet Table* content changed from t_1 to t_6 is displayed separately in Figure 3.5 to Figure 3.10. Table 3.1 shows the updated *AppSet Table* content at the end of t_6 . A dynamic *AppSet Table* updating operation at each scheduling interval is illustrated with the following figures. The red colored fonts indicate the updating operation happened to *AppSet Table* table every time the *Updater* Script is called by the *Monitor* script.

Figure 3.5 shows at t_1 , $n=1$ and $k=1$, the *Updater* creates a new entry in *AppSet Table* with *AppSet* name a1. The TSP is calculated based on the total power P_t and the background power P_b . The TIP for a1 is not available (NA) since there is no other application running in *AppSet*. The length of the interval $L(t_1)$ is 500 ms. The Unit Thread Size T_{size} for *AppSet*(a1) is recorded as 600 KB and the frequency of occurrence for *AppSet*(a1) is now 1.

AppSet	TSP (P_s)	TIP (P_d)	$P_t(W), L(ms)$	T_s, T_{size}	Range	Frequency
a1	$P_s(a1)=P_t(t1)-P_b=0.3$	NA ($P_b=3$, at the beginning of logging.)	$P_t(t1)=3.3$ $L(t1)=500$	$T_s(a1)=600$	NA	$F_s(a1)=1$

Figure 3.5 *AppSet Table* at t_1

Figure 3.6 shows at t_2 , $n=2$ and $k=1$, the *Updater* creates a new entry in *AppSet Table* with *AppSet* name a1,b1. The TSP is calculated based on the total power P_t and the TIP P_d monitored at the interval t_2 . The length of the interval $L(t_2)$ is also recorded. The Unit Thread Size T_{size} for *AppSet*(b1) is recorded as 700 KB. The frequency of the occurrence for *AppSet*(a1) and *AppSet*(b1) are both increased by 1 since they are subsets of *AppSet*(a1,b1). The *AppSet Table* only updates the occurrence of class 1 *AppSet* when their TSP is computable. At t_2 , only *AppSet*(a1,b1) and *AppSet*(a1) is logged into the *AppSet Table*, as *AppSet*(b1) is yet to be monitored alone by the scheduler prior to t_3 and its TSP is yet computable at t_2 .

AppSet	TSP (P_s)	TIP (P_d)	$P_t(W), L_t(ms)$	T_s, T_{size}	Range	Frequency
a1	$P_s(a1)=P_t(t1)-P_b=0.3$	NA	$P_t(t1)=3.3$ $L(t1)=500$	$T_s(a1)=600$	NA	$F_s(a1)=2$
a1,b1	$P_s(a1,b1) = P_t(t2)-P_d(t2)$	$P_d(a1,b1)=P_t(t2) -P_s(a1,b1)$	$P_t(t2)=4.3$ $L(t2)=300$	$T_s(b1)=700$	NA	$F_s(a1,b1)=1$

Figure 3.6 *AppSet Table* at t_2

In Figure 3.7, it is now at t_3 , $n=3$ and $k=1$, the *Updater* creates a new entry in *AppSet Table* with *AppSet* name b1. This is a quiet interval where the application b1 is the only running application. The TSP is calculated based on the total power $P_t(t_3)$ and the background power P_b . $P_s(b1)$ is needed for calculating $P_s(a1,b1)$. Upon acquiring the value of $P_s(b1)$, the *Updater* updates TIP for *AppSet*(a1,b1).

AppSet	TSP (P_s)	TIP (P_d)	$P_t(W), L(ms)$	$T_s, Tsize$	Range	Frequency
a1	$P_s(a1)=P_t(t1)-P_b=0.3$	NA	$P_t(t1)=3.3$ $L(t1)=500$	$T_s(a1)=600$	NA	$F_s(a1)=2$
a1,b1	$P_s(a1,b1)=P_t(t2)-P_d(t2)=0.4$	$P_d(a1,b1)=P_t(t2)-P_s(a1)-P_s(b1)-P_b=0.3$	$P_t(t2)=4.3$ $L(t2)=300$	NA	NA	$F_s(a1,b1)=1$
b1	$P_s(b1)=P_s(t3)-P_b=0.7$	NA	$P_t(t3)=3.7$ $L(t3)=100$	$T_s(b1)=700$	NA	$F_s(b1)=2$

Figure 3.7 AppSet Table at $t3$

The update at $t4$ is shown in Figure 3.8, where $n=3$ and $k=1$. The *Updater* did not create a new entry in *AppSet Table* since *AppSet(a1, b1)* exists. $P_t(t4)$ is now compared with $P_t(t2)$. $P_s(a1,b1)$ is recalculated based on the average value of the old $P_t(a1, b1)$ and the new $P_t(a1, b1)$ ' total power monitored. The difference between the updated value and the original value is then recorded under the Range column. Note that the red color *AppSet(a1, b1')* entry is not a new entry. It is the new total power of *AppSet(a1, b1)* collected at $t4$ and used to recalculate the TIP-TSP values in *AppSet(a1, b1)*.

AppSet	TSP (P_s)	TIP (P_d)	$P_t(W), L(ms)$	$T_s, Tsize$	Range	Frequency
a1	$P_s(a1)=P_t(t1)-P_b=0.3$	NA	$P_t(t1)=3.3$ $L(t1)=500$	$T_s(a1)=600$	NA	$F_s(a1)=3$
a1,b1	$P_s(a1,b1)=P_t(t2)-P_d(t2)=0.4$	$P_d(a1,b1)=P_t(t2)-P_s(a1)-P_s(b1)-P_b=0.3$	$P_t(t2)=4.3$ $L(t2)=300$	NA	0.05 Watt	$F_s(a1,b1)=2$
b1	$P_s(b1)=P_s(t3)-P_b=0.7$	NA	$P_t(t3)=3.7$ $L(t3)=100$	$T_s(b1)=700$	NA	$F_s(b1)=3$
a1,b1'	$P_{s'}(a1,b1)=P_s(a1,b1)-\frac{(P_t(t2)+P_t(t4))}{2}=4.25$	$P_{d'}(a1,b1)=P_t(t4)-P_{s'}(a1,b1)=0.25$	$P_t(t4)=4.2$ $L(t4)=100$	NA	0.05 Watt	$F_s(a1,b1)=2$

Figure 3.8 AppSet Table at $t4$

In Figure 3.9, where the interval is now $t5$, $n=4$ and $k=1$. The *Updater* creates a new entry in *AppSet Table* for *AppSet(a1, b1, c1)*. Since $P_s(c1)$ is unknown, the *Updater* can only record the corresponding power equations for now. The total power for $t5$ and the frequency are both updated.

AppSet	TSP (P_s)	TIP (P_d)	$P_t(W), L(ms)$	$T_s, Tsize$	Range	Frequency
a1	$P_s(a1)=P_t(t1)-P_b=0.3$	NA	$P_t(t1)=3.3$ $L(t1)=500$	$T_s(a1)=600$	NA	$F_s(a1)=4$
a1,b1	$P_s(a1,b1)=P_t(t2)-P_d(t2)=0.4$	$P_d(a1,b1)=P_t(t2)-P_s(a1)-P_s(b1)-P_b=0.3$	$P_t(t2)=4.3$ $L(t2)=300$	NA	0.05 Watt	$F_s(a1,b1)=3$
b1	$P_s(b1)=P_s(t3)-P_s(a1)=0.7$	NA	$P_t(t3)=3.7$ $L(t3)=100$	$T_s(b1)=700$	NA	$F_s(b1)=4$
a1,b1'	$P_{s'}(a1,b1)=P_s(a1,b1)-\frac{(P_t(t2)+P_t(t4))}{2}=4.25$	$P_{d'}(a1,b1)=P_t(t4)-P_{s'}(a1,b1)=0.25$	$P_t(t4)=4.2$ $L(t4)=100$	NA	0.05 Watt	$F_s(a1,b1)=3$
a1,b1,c1	$P_s(a1,b1,c1)=P_s(a1,b1)+P_s(c1)$	$P_d(a1,b1,c1)=P_t(t5)-P_s(a1,b1,c1)$	$P_t(t5)=5.2$ $L(t5)=100$	NA	NA	$F_s(a1,b1,c1)=1$

Figure 3.9 AppSet Table at $t5$

Figure 3.10 shows that at $t6$, $n=5$ and $k=1$, the *Updater* creates a new entry in *AppSet Table* for *AppSet(a1, c1)*. Since $P_s(c1)$ is not known yet, the *Updater* can record only the corresponding power

equations for now. Once $P_s(c1)$ is calculated at a quiet interval later, $P_s(a1, b1, c1)$ will be updated. The total power for $t6$ and the frequency are both updated.

AppSet	TSP (P_s)	TIP (P_d)	$P_t(W), L(ms)$	$T_s, Tsize$	Range	Frequency
a1	$P_s(a1)=P_t(t1)-P_b=0.3$	NA	$P_t(t1)=3.3$ $L(t1)=500$	$T_s(a1)=600$	NA	$F_s(a1)=5$
a1,b1	$P_s(a1,b1)=P_t(t2)-P_d(t2)=0.4$	$P_d(a1,b1)=P_t(t2)-P_s(a1)-P_s(b1)-P_b=0.3$	$P_t(t2)=4.3$ $L(t2)=300$	NA	0.05 Watt	$F_s(a1,b1)=3$
b1	$P_s(b1)=P_s(t3)-P_s(a1)=0.7$	NA	$P_t(t3)=3.7$ $L(t3)=100$	$T_s(b1)=700$	NA	$F_s(b1)=4$
a1,b1'	$P_{s'}(a1,b1)=P_s(a1,b1)-\{P_t(t2)+P_t(t4)\} \div 2=4.25$	$P_{d'}(a1,b1)=P_t(t4)-P_{s'}(a1,b1)=0.25$	$P_t(t4)=4.2$ $L(t4)=100$	NA	0.05 Watt	$F_s(a1,b1)=3$
a1,b1,c1	$P_s(a1,b1,c1)=P_s(a1,b1)+P_s(c1)$	$P_d(a1,b1,c1)=P_t(t5)-P_s(a1,b1,c1)$	$P_t(t5)=5.2$ $L(t5)=100$	$T_s(c1)=400$	NA	$F_s(a1,b1,c1)=1$
a1,c1	$P_s(a1,c1)=P_s(a1)+P_s(c1)$	$P_d(a1,c1)=P_t(t6)-P_s(a1,c1)$	$P_t(t6)=4.2$ $L(t6)=100$	NA	NA	$F_s(a1,c1)=1$

Figure 3.10 *AppSet Table* at $t6$

3.2.7 User Power Profile Recognition and Computation Space

AppSets are updated according to the scheduling intervals monitored. *AppSet Table* may not contain all the possible combinations of all the applications with all possible thread loads. The complete number of possible *AppSets* can be calculated with Equation (3.9),

$$\sum_{i=1}^{i=n} C_n^i \times k^i \quad (3.9)$$

where

n : the number of applications contained in *AppSet Table*, and

k : the number of the largest thread occurred at scheduling interval in the multiples of the average unit thread size.

As discussed earlier, general mobile users will frequently access 10 or less apps in their mobile devices. And the app usage patterns are often repetitive. The complexity of the *AppSets* monitored will be much less compared to the theoretical value calculated in (3.9). The usage pattern of a certain user can be represented in a series of percentage of *AppSets* used. The percentage is used to represent the total time of the *AppSet* that had its threads running in the past intervals. It is named the probability of presence, **Pr**. The probability of presence of an *AppSet* is calculated with several equations. At first the average interval length of the *AppSet* is calculated. A sample calculation of the average interval length $L_{average}(s)$ for *AppSet*(s) is shown in Equation (3.10).

$$L_{average}(s) = \frac{\sum_{i=1}^{i=n} L(i)}{F_s(s)}, \quad L(i) = \begin{cases} L(i), & AppSet(s) \in AppSet(i) \\ 0, & otherwise \end{cases} \quad (3.10)$$

where

$L_{average}(s)$: the average interval length of *AppSet*(s),

n : the sequence numbers of *AppSets* in *AppSet Table*,

$L(i)$: the interval length of $AppSet(i)$, and

$F_s(s)$: the times of occurrence of $AppSet(s)$.

The interval length values of the $AppSets$ that contains $AppSet(s)$ are summed to produce the total time of presence of $AppSet(s)$. Then the total time of presence of $AppSet(s)$ is divided by $F_s(s)$ to compute $L_{average}(s)$. The average length of an $AppSet(j)$ can be calculated at runtime with Equation (3.11).

$$L_{average}'(s) = \frac{L_{average}(s) \times [F_s(s) - 1] + L(j)}{F_s(s)} \quad (3.11)$$

where

$AppSet(j)$: the $AppSet$ present in the newly monitored scheduling interval,

$L_{average}(j)$: the interval length of the $AppSet(j)$, and

$L(j)$: the interval length of $AppSet(j)$.

The $L_{average}(s)$ will be updated with (3.10) when the $AppSet(s)$ is an element of $AppSet(j)$. After calculating the $L_{average}(s)$, the probability of presence of $AppSet(s)$ can be computed with Equation (3.12).

$$Pr(s) = \frac{L_{average}(s) \times F_s(s)}{T_m - T_0} \quad (3.12)$$

where

$Pr(s)$: the probability of presence of $AppSet(s)$,

T_m : the time stamp of the current scheduling interval, and

T_0 : the time stamp of the first scheduling interval monitored.

The probability is computed with the total time of the scheduling intervals of an $AppSet$ divided by the total time lapsed of the app usage pattern monitored. The total time is calculated by subtracting the time stamp of the first scheduling interval from the time stamp of the current scheduling interval. The probability of presence of an $AppSet$ will be used to predict the $AppSet$ at the next scheduling interval. Table 3.2 shows an example of the computed probability table based on the frequency values of the example given in Figure 3.10. For example, probability for $AppSet(a1)$ is calculated with $\frac{L(t1)+L(t2)+L(t4)+L(t5)+L(t6)}{L(t1)+L(t2)+L(t3)+L(t4)+L(t5)+L(t6)} = \frac{500}{600} = 83\%$ and that for $AppSet(a1, c1)$ is calculated with $\frac{L(t5)+L(t6)}{L(t1)+L(t2)+L(t3)+L(t4)+L(t5)+L(t6)} = \frac{200}{600} = 33\%$.

Table 3.2 The probability of presence table derived base on Table 3.1

<i>AppSets</i>	Probability of Presence
a ₁	83%
b ₁	67%
c ₁	33%
a ₁ , b ₁	50%
a ₁ , c ₁	33%
a ₁ , b ₁ , c ₁	17%

The probability of presence table shows the app usage pattern in a form of percentage. Base on the app usage pattern the application sequence of the scheduling queue can be predicted. The higher the probability is,

the more frequent the *AppSet* has appeared and the more likelihood it will appear again. In Section 3.2, the *Scheduler* script makes use of the probability of presence table to predict the likelihood of *AppSets* occurring at the next interval. The prediction is for the *Scheduler* script to decide the Scheduling *Scheme* for the current *AppSets* by considering the impact of future *AppSets*.

3.3 Application Categorization and Scheme Allocation

The user power profile consists of the *AppSet Table* and the probability of presence discussed in the previous section. The *AppSet Table* provides the *Optimizer* in the proposed power management approach with the power dissipation values of all possible multithreading situations. The *Optimizer* script uses the TIP values recorded in the *AppSet Table* to create the least TIP dissipated scheduling sequence, namely the optimized scheduling sequence. The probability of presence table is used to model the likelihood of next incoming or expiring thread. The *Scheduler* script will decide which scheduling scheme to be used based on the scheduling sequence provided by the *Optimizer* and will predict the future incoming *AppSets* using the probability of presence table. The native Android schedulers do not consider the difference of TIP component between different multithreading combinations (*AppSets*). The native schedulers schedule the user interactive applications threads according to the sequence of the thread arrival time. Figure 3.11 shows a native Android system under the governor of Completely Fair Scheduler (CFS) [150], which allocates the time slices for a series of multithreaded user interactive applications. In Android system, apps are scheduled by the underlying Linux kernel scheduler, and in Android 4.0 the Linux kernel used is Completely Fair Scheduler (CFS). In CFS, multi-threading among different applications are done using the time slices mechanism shown in Figure 3.11.

Figure 3.11 Equally distributed time slices at multithreading intervals

In the proposed power management approach, the order of the scheduling sequence of the *AppSets* is changed to reduce the TIP dissipated in a multithreading operation. The *Scheduler* script implements three proposed scheduling schemes to realize the optimized scheduling sequence provided by the *Optimizer*. These schemes make use of holding function and CPU affiliation function to change the sequence of the threads scheduled. The detailed scheme and *Scheduler* script are discussed in the following sections. In Android, apps

are normally given one event thread each. Thus, only the app sequence number is shown in Figure 3.11 instead of the thread numbers while discussing the scheduling of Android apps.

3.3.1 Power Management Schemes

The user pattern aware scheduling schemes are proposed to rearrange the scheduler queue for managing the Thread Interference Power. The *Scheduler* script chooses which scheme to apply based on the probability of presence and the optimized scheduling sequence provided by the *Optimizer* script. The three proposed scheduling schemes to be used by the scheduler are described below.

1. **HOLD scheme:** this scheme keeps other threads waiting in the scheduling queue while processing a single thread in a single core. Keeping the threads in scheduling queue meaning the threads will be stored in the DRAM for a longer period of time comparing to what it should be in the native multithreading scheme. As discussed previously, the TIP dissipation is the result of thread switching operation for multithreading. In Figure 3.12, the thread switching will not occur while the neighboring time slices are occupied by the same threads. Therefore this scheme will induce more Thread Solitary Power (TSP) dissipation and reduce the TIP dissipated at the same time. The *Scheduler* script will select this scheme in the situation that the TIP reduced is more than the TSP induced. The HOLD scheme delays execution of the app on hold for a brief moment, however the estimated completion time of the app is strictly followed to avoid system response time degradation.

Figure 3.12 Comparison between the Native Scheme and proposed HOLD scheme

2. **DEDICATED scheme:** this scheme schedules only one *AppSet* in a CPU core. It exploits the parallelism of multi-core systems to increase the computation throughput and reduce the TIP dissipation caused by the thread switching operations. As shown in Figure 3.13, the threads of an *AppSet* are scheduled into different CPU cores with threads of the same *app* in the same core. The TIP components are negligible since there is only one thread in each core. The TSP is kept to minimum as DRAM size required for thread queuing is small. The efficiency of this scheme depends on the balancing of the threads in different CPU cores. Unbalanced threads will cause idling of CPU cores and power will be dissipated unnecessarily. In Figure 3.13, the threads processed by Core 2 and Core 3 are shorter compared to the other 2 cores. New *AppSets* should be scheduled to Core 2 and Core 3

to avoid the idling of the CPU core. Accurate prediction of incoming *AppSets* will help the scheduler to keep CPU cores busy and the TIP dissipation minimized by planning the scheduling before the arrival of the *AppSets*. If the prediction is unreliable, the scheduler will have to switch off the cores for less power dissipation before the TIP saved can cover the power dissipated in switching on the extra cores.

Figure 3.13 An example of the DEDICATED Scheme

3. **TOTAL PARALLELISM** scheme: This *Scheme* is a combination of the DEDICATED Scheme and HOLD scheme. As shown in Figure 3.14, the optimized sequence of the current *AppSet* is balanced to different CPU cores to increase the computation throughput. Comparing to the HOLD scheme, this scheme dissipates more TIP due to frequent thread switching. On the other hand, as the throughput increases the TSP dissipations induced by this scheme are less comparing to the HOLD scheme. This scheme is used when the prediction accuracy is unclear and there are idling CPU cores to utilize.

Figure 3.14 An example of the TOTAL PARALLELISM Scheme

The *Scheduler* script will apply the three scheduling schemes depends on the system resources availabilities and the prediction accuracy of the future *AppSets*. The HOLD scheme will be applied when there is enough free scheduling time in a CPU core. The Parallelism scheme will be adopted when there is enough scheduling time in more than one CPU cores. The DEDICATED scheme will be utilized if the prediction is accurate and more than one CPU cores is free. The details of the decision making are discussed in the following sections.

3.3.2 Thread Interference Power and Thread Solitary Power Optimization

The application threads in the scheduler are organized based on the *AppSet Table* values to form optimized *AppSets* scheduling sequence that contributes the least TIP. The *Scheduler* reduces the overall power consumption by selecting scheduling schemes without degrading the system performance. In such case the relationship between the TIP caused by *AppSets* and the TSP caused by suspending the *AppSets* in the

scheduler have to be considered. The *Optimizer* script as shown in Script 4, will determine the optimized scheduling sequences of *AppSets* based on their TIP, TSP and average length of scheduled interval.

The TIP dissipation of an *AppSet* can be reduced by decomposing the *AppSet* into a combination of *AppSets* with lower TIP dissipations. The decomposed *AppSets* are scheduled sequentially to avoid TIP dissipation. This scheduling makes use of the HOLD scheme and the TSP aggregates for the *AppSets* scheduled later. Hence the *Optimizer* identifies the optimized sequence based on the total TIP and the total TSP dissipated of the decomposed *AppSets*. The total TIP of the *AppSets* decides which decomposed combination to choose and the TSP decides the sequence of the *AppSets* in the optimized scheduling sequence. To avoid the situation where the TSP components of an *AppSet* aggregate to overcome the TIP saved. A holding time limit is calculated for each holding operation. The holding of an *AppSet* will break when the timer expires.

SCRIPT 4 OPTIMIZER

```

1: Upon CALL Optimizer, receive Proc( $T_m$ ), at  $T_m$ 
2:   AppSet( $p$ )=AppSet(Proc( $T_m$ ))
3:   SEARCH AppSet( $p$ ), AppSet Table           #return all AppSets belong or equal
to AppSet( $p$ )
4:   DECOMPOSE AppSet( $p$ ) with AppSets returned #returns Sequence[1,2,..., $l$ ]
5:   SORT Sequence[1,2,..., $l$ ], according to  $P_d$ , ascending order #return the sorted Sequence Table
6:   SORT the AppSets in Sequence Table, according to  $P_s$ , descending order #return sorted Sequence( $o$ )
7:   return Sequence( $o$ )
8:   LENMAX Sequence( $o$ )                       # Calculate the maximum length for
Sequence( $o$ )
9:   IF  $L_{max}(o) < L_{th}(o)$ 
9:     return  $L_{max}(o)$ 
11:  ELSE
12:    return  $L_{th}(o)$ 
13:  END IF

```

where

AppSet(p): the *AppSet* containing all application IDs at current interval T_m ,

SEARCH: search and return *AppSets* belong or equal to *AppSet*(p) in the *AppSet Table*,

DECOMPOSE: arrange possible combinations of *AppSets* returned by SEARCH to reassemble *AppSet*(p),

Sequence[1,2,..., l]: the *Sequence table* contains all possible *AppSets* reassemble *AppSet*(p),

SORT: sort the data group according to the selected parameter in descending or ascending order,

o : the sequence number of the *Sequence* with the least P_d in the *Sequence table*,

Sequence(o): the optimized scheduling sequence of the *AppSet*(p),

LENMAX: calculates the maximum length values for *Sequence*(o),

$L_{max}(o)$: the maximum scheduling interval length of *Sequence*(o), and

$L_{th}(o)$: the threshold length of *AppSet* can be scheduled without causing system degradation.

An example of the *Optimizer* script sorting the optimized sequence is given below. In this example the process waiting in the scheduler queue is shown in Figure 3.15, where *AppSet*(*Proc*(T_m)) = *AppSet*($a1, b1, c1$).

 Figure 3.15 The scheduler queue at T_m

The *AppSet Table* used in this example is the same as Table 3.1. *AppSets* in Table 3.1 are used as the basic *AppSet* to make scheduling decision for application threads shown in Figure 3.15. According to the *AppSets* logged in Table 3.1, there are four combinations of *AppSets* that contains thread a1, b1 and c1. The combinations are then sort according to their TIP. The SEARCH function returns all *AppSets* in Table 3.1 belong to *AppSet*(a1, b1, c1). The DECOMPOSED function combines the returned *AppSets* to establish the *Sequence* table. Then the SORT function reorders the *AppSets* combinations according to their TIP values as shown in Table 3.3. There is no combination of *AppSet*(b1, c1) + *AppSet*(a1) since *AppSet*(b1, c1) does not exist in the *AppSet Table*. The *AppSets* in Table 3.1 are used as basic *AppSets* to make scheduling decisions for application threads shown in Figure 3.15. According to *AppSets* logged in Table 3.1, there are 4 combinations of *AppSets* that contains a1, b1 and c1. The combinations are then sorted according to their TIP.

Table 3.3 Sequence Table derived from Table 3.1 sorted according to TIP

Sequence number	<i>AppSets</i> Combinations	TIP (W)
1	<i>AppSet</i> (a1) + <i>AppSet</i> (b1) + <i>AppSet</i> (c1)	0
2	<i>AppSet</i> (a1, b1) + <i>AppSet</i> (c1)	0.3
3	<i>AppSet</i> (a1, c1) + <i>AppSet</i> (b1)	0.5
4	<i>AppSet</i> (a1, b1, c1)	0.8

After determining the optimized *AppSet* content based on TIP, the scheduling sequence inside the optimized *AppSet* is determined by the TSP value of each member *AppSet*. The member *AppSets* with larger TSP get to be scheduled sooner to avoid excessive TSP dissipation. As shown in Table 3.4, by calling the SORT function again scheduling sequence of the *AppSets* in each combination will be arranged according to their TSP values in descending order. The *AppSets* with larger TSP dissipation will be scheduled first. The “>” connecting *AppSets* indicates the amount of the TSP dissipated by the *AppSets* and the scheduling sequence. The *AppSets* on the left side of “>” dissipate more TSP than the *AppSets* on the right side. The *AppSets* on the left will be scheduled before the *AppSets* on the right. Putting apps with larger TSP on hold results in higher TSP dissipation over time, hence apps with larger TSP always gets scheduled first. As shown in Table 3.4, the optimized scheduling sequence *Sequence(o)* for the *Proc*(T_m) is *AppSet*(b1) > *AppSet*(c1) > *AppSet*(a1).

 Table 3.4 *AppSets* combinations scheduling sequence rearranged according to TSP

Sequence number	<i>AppSets</i> Combinations	TSP (W)
1	<i>AppSet</i> (b1) > <i>AppSet</i> (c1) > <i>AppSet</i> (a1)	$P_s(a1)=0.3$ $P_s(b1)=0.7$ $P_s(c1)=0.4$
2	<i>AppSet</i> (a1, b1) > <i>AppSet</i> (c1)	
3	<i>AppSet</i> (a1, c1) > <i>AppSet</i> (b1)	
4	<i>AppSet</i> (a1, b1, c1)	

After determining the *AppSet* content based on optimized TIP, the scheduling sequence inside the optimized *AppSet* is determined by the TSP value of each member *AppSet*. The member *AppSets* with larger TSP are scheduled sooner to avoid excessive TSP dissipation. As discussed previously, the optimized sequence is implemented with the HOLD scheme. The TSP dissipated by the threads on hold will increase as the holding time increases. The maximum length of the scheduling interval the optimized sequence can be allocated without causing too much TSP dissipation is derived with the model illustrated in Figure 3.16.

Figure 3.16 The TIP and TSP in different *AppSets* scheduling sequences

In Figure 3.16, $L(a)$ and $L(b)$ indicate respectively the length of the scheduling intervals of *AppSet(a)* and *AppSet(b)*. The *AppSet(a,b)* is the scheduling sequence of the *Native* Android scheduling scheme. It has larger amount of TIP dissipation ($P_d(a,b)$) and less TSP dissipation since all threads are multithreaded evenly. The *AppSet(a) > AppSet(b)* sequence generates negligible TIP. The TSP dissipated by *AppSet(b)* is larger due to the holding of the scheduling. In an effective scheduling the total energy consumed by the sequence *AppSet(a) > AppSet(b)* is less than the sequence *AppSet(a,b)*. To achieve this optimum condition, the scheduling interval length of *AppSet(a)* cannot exceed a limit named the maximum scheduling length. The maximum scheduling length defines the maximum length the holding *AppSet* can wait without increasing the total energy dissipated. To deduce the maximum length for the *AppSet(a)*, assume that the scheduling interval length of *AppSet(b)* is the average interval length, $L(b) = L_{average}(b)$ and let $L(a) = L_{max}(a)$. Then the energy equations for *AppSet(a,b)* and *AppSet(a) > AppSet(b)* are formed and their relation is given in (3.13). After simplification, the maximum length of scheduling $L_{max}(a)$ can be derived with (3.14).

$$L_{max}(a) \times P_s(a) + [L_{max}(a) + L_{average}(b)] \times P_s(b) < [L_{max}(a) + L_{average}(b)] \times P_d(a,b) + L_{max}(a) \times P_s(a) + L_{average}(b) \times P_s(b) \quad (3.13)$$

$$L_{max}(a) < \frac{L_{average}(b) \times P_d(a,b)}{P_s(b) - P_d(a,b)} \quad (3.14)$$

To generalize Equation (3.14), the parameters related to a will have the values of the running *AppSet* and the parameters related to b have the values of the holding *AppSet*.

There are three expiration conditions for scheduling interval of an optimized sequence scheduled by the *Scheduler*.

1. The thread scheduling interval expires normally.
2. The thread exceeds the 200ms interval limit. This is to prevent a thread occupying the CPU core for too long and defeating the purpose of multithreading. The limit can be dynamically changed to avoid performance degradation.
3. The thread exceeds the maximum scheduling length.

In this example, $L_{max}(b1)$ and $L_{max}(c1)$ are calculated, giving $L_{max}(b1) = 600ms$ and $L_{max}(c1) = 550ms$. The maximum length is longer than the 200ms interval limit. $AppSet(a1)$, $AppSet(b1)$ and $AppSet(c1)$ will all be given 200ms scheduling interval by the *Scheduler*. The interval will be expired and switch to the next *AppSet* in queue if the current *AppSet* cannot finish its task in 200ms. The default time slice length in the native Linux Completely Fair Scheduler (CFS) is 100 milliseconds. The initial interval limit is set as two times of the default value to avoid notable latency in application execution.

3.3.3 Scenario Decision

The *Scheduler* script decides which power management scheme to apply based on the optimized scheduling sequence provided by the *Optimizer*, the available system resources and the prediction derived from the probability of presence table. The detailed decision making policy is given in Script 5.

SCRIPT 5 SCHEDULER

```

1: Upon thread scheduling events is true,  $e = 1$  at instance  $T_m$ 
2: WHILE  $e = 1$ 
3:     PREDICT  $Proc(T_{m+1})$ ,                                #determine events at next interval, return the
Pred(Tm+1)
4:     CALL Optimizer,  $Proc(T_m)$ ,                          #analyze  $Proc(T_m)$ , return Sequence(o) and
Lmax(o)
5:     IF there is enough idling CPU time to fulfill the  $L_{max}(o)$  # enough free CPU resources to schedule
Sequence(o)
6:         IF there is free CPU core                            #idling CPU cores to schedule Sequence(o) in
parallel
7:             IF  $Pred(T_m) = Proc(T_m)$                         # the previous prediction is
accurate.
8:                 SCHEDULE  $Proc(T_m)$ , Dedicated Scheme
9:             ELSE
10:                SCHEDULE  $Proc(T_m)$ , Total Parallelism Scheme
11:            END IF
12:        ELSE
13:            SCHEDULE  $Proc(T_m)$ , Hold Scheme
14:        END IF
15:    ELSE
16:        SCHEDULE  $Proc(T_m)$ , Native Scheme
17: END WHILE

```

where,

PREDICT: predicts the *AppSets* at given interval by going through the probability of presence table,

$Proc(T_{m+1})$: indicates the PREDICT function to predict the *AppSets* at T_{m+1} ,

$Pred(T_{m+1})$: the predicted *AppSets* at interval T_{m+1} ,

$L_{max}(o)$: the maximum length values of $Sequence(o)$, and

SCHEDULE: schedule the *AppSets* with the selected *Schemes*.

In the HOLD scheme the time slices of one thread are grouped together to form a consecutive time slice named the SUPER slice. As shown in Figure 3.17 the SUPER slice is realized by dividing the interval T_m into multiple sub-intervals. In order to schedule an *AppSet* with the HOLD scheme, the scheduler will set the expiration time according to the $L_{max}(o)$ or the $L_{th}(o)$ for the scheduled *AppSet*. Once the expiration is reached, the scheduler will suspend the running *AppSet* and schedule the next *AppSet* likewise. This procedure will repeat until one of the *AppSets* is forced to retire by the user or the system. The scheduled *AppSets* will be reanalyzed by the *Optimizer* upon new scheduling event. Then the scheduling sequence will change according to the new $Sequence(o)$.

Figure 3.17 The Native and HOLD scheme

In the TOTAL PARALLELISM scheme the SUPER slice allocation is realized with the same approach used by the HOLD scheme. As shown in Figure 3.18, the TOTAL PARALLELISM scheme will separate the *AppSets* into equal groups and stream into idling CPU cores. This is achieved by assigning different CPU core affiliation to each *AppSet* group. The scheme will be changed to HOLD scheme from TOTAL PARALLELISM scheme when CPU cores are allocated to other *AppSets* or switched off by the system.

Figure 3.18 The realization of Parallelism Schemes

The DEDICATED scheme requires accurate *AppSets* prediction. The *AppSets* prediction is achieved with the probability of presence table with the method described below. The *AppSets* in the probability of presence table are given classes equal to their member applications. The *AppSets* with only one application are the Class 1 *AppSets*, the *AppSets* contain two applications are the Class 2 *AppSets*, and so on. The presence of an *AppSet(s)* at any interval is determined by (3.15),

$$P_p(s) = \frac{L_{average}(s) \times F_s(s) - L_{sh}(s)}{T_m - T_p} \quad L_{sh}(s) = \begin{cases} L_{sh}(s) + T_p, & \text{AppSet}(s) \text{ is present at } T_p \\ L_{sh}(s), & \text{otherwise} \end{cases} \quad (3.15)$$

where

$P_p(s)$: the probability of the *AppSet* occurs in the current interval, not equal to $Pr(s)$,

$L_{sh}(s)$: the total length of time of *AppSet*(s) being scheduled, and

T_p : the total length of time passed since the beginning of the prediction.

Figure 3.19 shows an example of predicting the *AppSets* presented at different intervals. The length of the intervals is assumed to be equal. Each row represents a scheduling interval. The percentage on the top left indicates the P_p of the *AppSets*. A threshold probability P_{th} value is used to determine the presence of an *AppSet* at a given interval. An *AppSet* will be determined as present in an interval if its P_p value is greater than the P_{th} . The P_{th} is determined dynamically to calibrate the prediction accuracy. In this example $P_{th}=75\%$. As computed with Equation (3.15), the P_p value of an *AppSet* will decrease if scheduled in the current interval. The P_p value will increase if the *AppSet* is absent from the interval. The circles with black numbers indicate that the *AppSets* are present in the interval. The circles with red numbers indicate the *AppSets* are absent in the interval. The circles with green numbers indicate the presence of the *AppSets* in the interval cannot be determined based on Class 1 *AppSets* alone. In such case the P_p of the relating *AppSets* with higher Class are considered to make the decision. Once the P_p of an *AppSet* decreases to a value smaller than P_{th} , the P_p of Class 2 *AppSets* will be checked. As shown in the row of T_3 of Figure 3.18, P_p of *AppSet*(2) dropped below the P_{th} . In order to determine the presence of *AppSet*(2), the P_p of *AppSet*(4) is checked against the product of P_p of *AppSet*(1) and $AppSet(2).$

T_p	Class 1 AppSets			Class 2 AppSets		Class 3 AppSets
T_0	83% 1	67% 2	33% 3	50% 4	33% 5	17% 6
T_1	80% 1	80% 2	40% 3	60% 4	40% 5	20% 6
T_2	75% 1	75% 2	50% 3	50% 4	50% 5	25% 6
T_3	100% 1	67% 2	67% 3	67% 4	67% 5	33% 6
T_4	100% 1	50% 2	100% 3	50% 4	100% 5	50% 6
T_5	100% 1		100% 3		100% 5	

1

2

3

4

5

6

1 AppSet(a1)

2 AppSet(b1)

3 AppSet(c1)

4 AppSet(a1,b1)

5 AppSet(a1,c1)

6 AppSet(a1,,b1c1)

Figure 3.19 An example of the prediction computation

The $Pred(T_{m+1})$ in Script 4 is predicted with the same approach. If the prediction is accurate, the *Scheduler* will schedule the given *Sequence(o)* with DEDICATED scheme. The predicted *AppSets* will be scheduled to CPU cores before coming to the scheduler queue. If the prediction is not accurate, the DEDICATED scheme will be switched to the TOTAL PARALLELISM scheme. At the same time, the probability of presence table will be updated with new values and new P_{th} increases the accuracy of the prediction in the future.

3.4 System Simulation and Result Verification

The feasibility of the proposed schemes was testified in a simulated environment. Several virtual users with different usage habits were modeled. Each user was represented with a different combination of applications and the usage of each application varies in each interval. Each combination was simulated for 12 intervals with equal length. A 12-interval simulation is termed a scenario. The *AppSet Table* and the probability of presence tables of different virtual user scenarios were then established at the end of each scenarios simulated. The scheduling scheme to reduce the Thread Interference Power (TIP) at each interval was determined based on the *AppSets* table and the probability of presence table. The scenarios for each virtual user were simulated for the three setups below.

1. The scenarios were simulated without applying any proposed schemes, i.e., the Native schemes of Android operating system is in charge of the scheduling.
2. The scenarios were simulated with the TOTAL PARALLELISM scheme applied for all intervals.
3. The scenarios were simulated with all the three proposed schemes applied for all intervals.

The resulting total power dissipation of the simulations with the proposed scheduling schemes applied, i.e. the second and third scenarios, was computed and compared with the simulation results with the Android Native scheme. The results are presented and discussed. The rest of the section is organized as follows. Section 4.1 discusses the system model of the simulation environment. In Section 4.2 the composition of virtual users are detailed. Section 4.3 illustrates the interval and *AppSets* allocation of a virtual user with the Native and the proposed schemes applied. In Section 4.4 the results of the simulations are presented and discussed.

3.4.1 System Model

The simulator used in the simulation is the GEM 5 simulator [96]. It is a Linux based simulator capable of simulating CPU event traces with consideration of the operating system. As shown in Figure 3.20 the operating system simulated by GEM 5 is Android ICS. The simulated processor is an ARM cortex A9, 2 GHz, quad core processor fabricated with 32nm technology. The simulated cores are running in simple order mode with 256 MB of RAM. L1 cache size is 32KB and the L2 cache size is 512KB.

Figure 3.20 The established GEM5 setup

In order to monitor the overall power dissipated by the simulated system, the McPAT power profile generator [97] was used in the simulation. The McPAT is capable to generating power profile of a processor based on the runtime CPU event traces. As shown in Figure 3.21, in this setup the event traces are generated by the GEM5 simulator and stored in the output files named Stats.txt and Config.py. The linker file named m5-mcpat.pl (GEM5 to McPAT Perl), converts these two output files into GEM5.xml. The McPAT reads the event traces from the GEM5.xml and computes the overall power consumption of the system simulated. In the simulation the event traces generated in every interval are collected and passed to the McPAT. The output power from McPAT will be logged into the *AppSets* table as the total power dissipation P_t .

Figure 3.21 The GEM5 and McPAT interface setup

The simulated scheduling intervals are modeled with the GEM .rcS script [41]. The simulation will suspend upon calling the .rcS checkpoint function at the end of each interval. The *AppSets* will be changed according to the scenarios. The scheduling *schemes* are modeled with the .rcS functions as well. To apply the HOLD scheme, an interval is separated into sub-intervals by calling the checkpoint function multiple times within an interval. The application scheduled in each sub-interval will rotate according to the given *Sequence(o)*. The TOTAL PARALLELISM scheme is modeled by creating identical thread sequences with shorter sub-intervals and each sequence will be issued with CPU affiliation to a different core. The DEDICATED scheme is modeled by assigning different CPU cores to different applications in the *Sequence(o)*.

3.4.2 Virtual Users and Simulation Scenarios

The virtual users with different application usage are simulated with Moby benchmark [102], which consists 10 applications and is capable of simulating scenarios of executing various applications together. In Moby the application running sequence is controlled by the .rcS script functions. The application sequences can be suspended by calling the checkpoint function to simulate the end of a scheduling interval. While the simulation is suspended, the GEM5 output files will be dumped and converted by the McPAT script to the total power dissipation of the interval. The total power dissipated in the interval will be logged as P_i for the *AppSet Table*. The executing applications will be changed during the suspension to simulate the changing of *AppSets* between intervals.

The simulation consists of different virtual user profiles. Each virtual user has an affiliated set of mobile Apps. This creates distinctive *AppSets* sequences to test the scheduler schemes against different user power profiles. As shown in Table 3.5, there are three virtual users, the General user, the Business user and the Media user. The General user will make use of all the apps with evenly distributed weightage except the Adobe reader and the Office. The Business user spends most of his time using business related apps. Unlike the General user, the Business user does not use the social apps. The Media user dedicates his time mainly with the multimedia apps and browses web page occasionally. The simulation *AppSets* sequence for each virtual user is determined based on the weightage values in Table 3.5.

Table 3.5 App Weightage Table for Different Virtual Users

	eBay	Adobe	Web	Email	Office	Map	News	Weibo	Mp3	Game
General	9.2%	0.00%	16.09%	8.05%	0.00%	12.64%	12.64%	11.5%	21.83%	8.05%
Business	3.33%	30.00%	20.00%	16.67%	20.00%	0.00%	0.00%	0.00%	10.00%	0.00%
Media	0.00%	0.00%	15.63%	12.5%	0.00%	0.00%	12.5%	12.5%	31.24%	15.63%

During the simulation, the time that an app is picked to run is based on the weightage in Table V. The total runtime of an app is given by Equation (3.16).

$$T_U(a) = W(a) \times T_U(o) \quad (3.16)$$

where,

$T_U(a)$: the total run time of the app a is up and running, including background time,

$W(a)$: the weightage of app a , and

$T_U(o)$: the summation of running and background time of all user interactive apps.

3.4.3 Simulation Procedures

The scenarios of different virtual users with the Android Native Scheme were simulated first. Figure 3.22 illustrates an example of the Business User scheduled with the Native Scheme. The *AppSets* scheduling intervals are derived according to the app weightage in Table 3.5. The apps are given application IDs from (a) to (f) for the purpose of indicating composition of *AppSets* in Figure 3.22. The number following the

application ID represents the number of threads issued by the app. For example b_2 is the second thread issued by the app Adobe. The *AppSets* table and the probability of presence table of the scenario are established with the *AppSets* monitored in each interval and the McPAT computed total power dissipation P_t .

Figure 3.22 The Business User Native Scheme scheduling scenario

Once the tables are updated with all *AppSets* data of the simulated *AppSet* sequence, the optimized scheduling *Scheme* will be determined for each interval based on the method presented in Section 3.2.1. The simulation for a virtual user was performed with TOTAL PARALLELISM scheme and repeated with All Schemes. The example of the Business user *All Schemes* simulation scenario is shown in Figure 3.23, where the colored bars indicate the sub-intervals created by the scheduling schemes. The symbols *Core 0* to *Core 3* indicate the affiliated CPU cores for each thread in the TOTAL PARALLELISM and DEDICATED scheme.

Figure 3.23 The Business User All Schemes scheduling scenario

3.4.4 Results and Discussion

The average power dissipation of the simulated scenarios is presented in Table 3.6. From the results, reductions of overall power dissipations are observed. Comparing the Native Scheme scenario to the All Scheme scenario, there is 19% of reduction of overall power consumption for the general user, 9% reduction for the business user and 6.5% reduction for the media user. The TOTAL PARALLELISM scheme did not show much improvement in the power saving. In order to investigate the reason of the underperforming TOTAL PARALLELISM scheme, the Native Scheme P_t values are decomposed into the TSP component P_s and the TIP component P_d with the background power P_b removed. Then the P_t values of the TOTAL PARALLELISM and All scheme are plotted against time with the decomposed Native Scheme values for each virtual user.

Table 3.6 The Simulation result

Virtual User	Native Scheme	Total parallelism Scheme	All Scheme	Power Reduction
General	3.37228 W	3.589 W	2.73 W	19%
Business	2.9202 W	2.9163 W	2.6412 W	9.6%
Media	3.2928 W	3.2553 W	3.0841 W	6.4%

As shown in Figure 3.24, Figure 3.25 and Figure 3.26, the P_t values of the All Schemes scenario are positively related to the TSP component, and negatively related to the TIP component most of the time. This indicates the combinations of the adopted scheduling schemes are efficient in reducing the TIP components occurred in multithreading operations. In Figure 3.24, the TOTAL PARALLELISM scheme demonstrated negative relationship with the TSP components. This indicates that the TOTAL PARALLELISM scheme is capable of reducing the TSP component by scheduling the identical *AppSets* into different CPU cores. But the TIP dissipation increases as the SUPER slice length decreases. Applying the DEDICATED scheme will solve this problem without reducing the computation throughput by increasing SUPER slice length. Hence it is important to have an accurate incoming thread predictor module. As shown in Figure 3.19 the user pattern based prediction is capable of deliver firm accurate prediction 75% of the time based on the probability of presence of Class 1 *AppSets* alone. The 75% prediction accuracy is deduced from the lower bound of the Class 1 *AppSets* prediction accuracy using the same prediction computation method used in the computation of Figure 3.19.

Figure 3.24 TIP and TSP of General user

Figure 3.25 TIP and TSP of Media user

Figure 3.26 Total power dissipated over time of Business user

Figure 3.27, Figure 3.28 and Figure 3.29 illustrate detailed TIP-TSP dissipated by each app for respective user group. The top chart in each figure shows the TIP recorded during the simulation with a sampling interval of 1500ms. The vertical axis denotes the amount of power dissipated by the apps in watts over each sampling interval. The power dissipation generated by each participating app is recorded and accumulated from the bottom following the order of the legends shown in the figure from left to right. The bottom chart in each figure shows the TSP recorded and the data is presented in the same layout as the TIP chart. From the detailed results it is observed that the TIP and TSP dissipation are not always in phase with each other. For example in Figure 3.27 the General user result from the intervals 41 to 50 and in Figure 3.29 the Business user result from the intervals 91 to 100, the TIP dissipation shows negative correlation with the TSP dissipation. Such behavior matches with one of the primary hypothesis of the TIP-TSP model, that the amount of TIP dissipation is not fully correlated to the TSP dissipation. The TIP component is more of an additional power dissipated through multithreading arrangement between different application threads and it varies according to the content of *AppSet* under execution.

Figure 3.27 TIP-TSP dissipation by app over time of General user

From another example shown in the sampling interval from 41 to 50 and 81 to 90 of Media user in Figure 3.28, it is observed that TIP dissipation is absent even there is a remarkable amount of TSP dissipation presented. This is due to the application threads in the given *AppSet* are well distributed to different CPU cores and scheduled in a way that *AppSet* with high TIP components are isolated from each other. By analyzing the detailed TIP-TSP result of the simulation, the TIP-TSP model assumptions are noticed to be valid. However it is also observed that the TIP-TSP relationship is difficult to predict without a mechanism to monitor performance of each individual application at runtime. Thus in Chapter 4, a performance to power dissipation quantization model is developed to meet the need of analyzing TIP-TSP components at runtime.

Figure 3.28 TIP-TSP dissipation by app over time of Media user

Figure 3.29 TIP-TSP dissipation by app over time of Business user

3.5 Conclusion and Further Works

The dark silicon effect had shifted the focus of mobile system design from increasing the number of cores to developing more conservative power management method. The power dissipation can be reduced with hardware implementation to lower the operating voltage of CPU cores and software intervention to manage the Thread Interference Power (TIP) dissipation in a system.

In this chapter the challenges faced by the prediction based management method have been discussed. The prediction based management methods tend to have lower accuracy in long term prediction. Hence to increase the accuracy of the prediction the usage pattern based prediction method was presented. The proposed self-calibrating prediction method had shown its capability of computing the future application up to the length of monitoring in Figure 3.19. As the length of monitoring increases the number of predictable *AppSets* will increase. In the proposed power management approach, the TIP components of applications are reduced by the scheduler. As shown in the simulated result, the prediction based DEDICATED scheme is more efficient compare to the others. It shows averagely 33% more TIP saving compared to the TOTAL PARALLELISM scheme. Increasing the accuracy of prediction and evaluating the performance of the proposed scheme is the prime focus next.

The proposed prediction based scheme is currently depending on the historical record of the application usage data (usage patterns). In other words there is a limitation that the scheme is unable to predict the future *AppSets*, if it contains *AppSets* that are not in the *AppSet Table*. In the following chapters this problem is solved by adopting the Artificial Neural Networks method that enables the prediction of possible unknown *AppSets* data based on the current *AppSet Table* via deep learning.

Another improvement that can enhance the current scheme is to incorporate DVFS. The DVFS is a widely adopted technique by the CPU cores to reduce power dissipation by adjusting the frequency and voltage pair at runtime. The DVFS can be incorporated into the currently proposed scheme to optimize the power dissipation of CUP cores as shown in Figure 3.30.

Figure 3.30 Proposed scheduling schemes with DVFS incorporated

In Figure 3.30, the DVFS controller can operate closely with the *Optimizer* and be in charge of computing the optimized frequency for the scheduling interval. This optimization of voltage and frequency pair can be based on the user power profile monitored and the optimized scheduling sequence computed. With DVFS, the performance of schemes proposed should be further enhanced. For example in the scenario where the DEDICATED scheme is applied and the prediction is only accurate for a brief scheduling interval, the CPU can be scaled to lower frequency to lower the throughput and reduce power dissipation so that the DEDICATED scheme can be applied with shorter period of prediction without degrading the performance. The proposed scheme can also be improved by exploiting the characteristic of heterogeneous CPU. The current scheme is based on symmetrical CPU designs where the power dissipation and clock frequency are identical in each CPU core. By considering heterogeneous CPU designs, the scheduler can choose to schedule *AppSets* with smaller sub-intervals into the core with lower frequency to reduce TIP dissipation, and reserve the CPU cores with high frequency for *AppSets* allocated with large sub-intervals to increase the throughput and reduce the TSP TIP caused by the *AppSets* suspended. Furthermore, the thermal dissipation in CPU core is not uniform. Some of the cores can dissipate more thermal energy compare to the rest. By considering the thermal dissipation of *AppSets* at each CPU, the proposed scheme can avoid scheduling *AppSets* with higher thermal dissipation to overheated CPU cores. The DVFS incorporation to improve the performance of the proposed scheduling schemes together with the consideration of thermal dissipation in different CPU cores is described in more details in the next chapter.

Chapter 4 App Usage Pattern Aware Linux Scheduler

4.1 Introduction

The application of simultaneous multithreading technique in Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) multiprocessor platforms has improved computing capacity of modern electronic devices for decades. However recently the computation capacity growth slows down and numerous power dissipation problems surface as the power wall problem gets serious. Research works introduce hardware based solutions such as new heterogeneous architectures and accelerators [103] [104] to overcome the barrier in symmetrical chip multiprocessor (CMP) architecture. In contrast to the hardware based approaches, software based solutions such as new scheduling approaches, migration techniques and thermal management algorithms [105] [106] have also proven their value in power reduction and gained support among the researchers due to its ease of implementation across different environments.

In the software based user pattern aware scheduling schemes proposed in Chapter 3, different usage patterns of users are identified and specific usage pattern based scheduling schemes are applied to reduce the overall power dissipation. Comparing to the other software based power management methods, the scheduling schemes proposed in Chapter 3 address the impact of multithreading different applications on power dissipation and have achieved better power reduction over the existing scheduling techniques. The fundamental concepts behind proposed scheduling schemes arrives from the observation that in the same group of apps, the sum of power consumption of apps executed individually is less than the power dissipated by the apps executed concurrently. The excessive amount of the power dissipated amongst concurrently running apps is named as the Thread Interference Power (TIP). The amount of TIP is unique to a group of apps and the groups are named as *AppSets*.

The work presented in this chapter is based on TIP-TSP model proposed in Chapter 3. In Chapter 3 three scheduling schemes are proposed, the DEDICATED, HOLD and TOTAL PARALLELISM, where TOTAL PARALLELISM is the combination of DEDICATED and HOLD scheme. The mechanism of TOTAL PARALLELISM is modified and renamed as SUPER in this chapter, which mitigates the TIP components by reducing the number of context switches with extended the time slice of a given application thread. Thus there are three basic scheduling schemes used in the context of this chapter, HOLD, SUPER and DEDICATED. All three schemes can be used in parallel as a combination or individually according to the Power Dissipation Problems (PDPs) identified and discussed in later sections. Comparing to Chapter 3 where only 1 combined scheduling scheme the TOTAL PARALLELISM scheme is available, the Re Scheduler proposed in this chapter has three individual scheduling schemes and is capable of applying more combined scheduling schemes such as the HOLD/DEDICATED, HOLD/SUPER, HOLD/SUPER/DEDICATED, etc.. Here, the mechanism of all three basic scheduling schemes are introduced individually and the effect of solving PDPs with scheduling schemes in parallel and individually are presented.

The study in Chapter 3 is based on Android. Android is based on Linux, hence scheduling decisions made in Android are passed into the Linux kernel underneath for execution. Thus directly applying scheduling

decisions in Linux kernel will increase the accuracy of gauging the system performance and reducing scheme application overheads. In Chapter 3 the scheduling schemes are realized using simulation scripts that controls Android app execution sequence while implementing the scheduling schemes Linux enables the incorporation of DVFS and thermal awareness in making the scheduling decisions.

The existing load balancing and scheduling mechanism of Linux operating system does not analyze this power dissipation pattern of TIP. Thus in this chapter a TIP-aware Linux Scheduler is developed based on the assumption of TIP and discusses the advantage of minimizing the TIP in multithreading Apps and maintaining CPU utilization at the same time by analyzing the TIP difference of each *AppSet*.

Heterogeneous architecture becomes more popular as an alternative to tackle the power and thermal wall in the current trend of processor development. From the study in Chapter 3, it is observed that the amount of TIP increases as the complexity to make multithreading decisions for app or application threads increases. In a heterogeneous processor the application threads have more degrees of freedom as there are different kinds of cores allocated in different CPU Cluster. Hence the amount of TIP generated during the complex process of scheduling decision making is expected to be more than that in a homogeneous processor.

In this chapter, a processor with three CPU Clusters is used to exploit the power efficiency that has been achieved by the existing two clustered heterogeneous processor like the big.LITTLE [66]. The proposed Linux Scheduler is deployed to compare its efficiency with the native Linux Schedulers in the heterogeneous processor.

The main contribution of this chapter includes:

- A Performance Metric Parameter (PMP) based quantitative model is formulated to monitor and analyze TIP dissipations patterns at runtime.
- Methods are proposed to identify Power Dissipation Problems (PDPs) based on the system performance metrics.
- DVFS is incorporated in the existing scheduling schemes for better power dissipation reduction.
- Thermal hot spot awareness is included in the scheduling scheme application.
- The proposed TIP analysis approach and scheduling schemes are included in a newly developed Linux-based scheduler, named as the Re Scheduler.
- The efficiency of the proposed Re Scheduler is assessed in an asymmetrical processor with three CPU Clusters namely Small.Medium.Big (S.M.B), and the results are compared with the Native Global Task Scheduler (GTS) [107] and the In Kernel Switcher (IKS) schedulers [68].

The proposed Re Scheduler is deployed in Linux Kernel 3.10 and 3.14, and tested in GEM5 [96] simulator using Moby benchmark [102]. The experiment results have shown that the Re Scheduler achieves a better power efficiency of up to 15% when comparing to the GTS and IKS in the three clustered processors.

The rest of this chapter is organized as follow. Section 4.2 presents the quantitative Thread Interference Power and Thread Solitary Power (TIP-TSP) modeling approach and the Power Dissipation Problem (PDP) to Scheduling Scheme Mapping. Section 4.3 illustrates the detailed scheduling scheme decision making process based on the TIP-TSP components monitored. Section 4.4 presents the setup of the multi clustered S.M.B

processor and the simulation environment. Section 4.5 presents the results and the discussions. Section 4.6 concludes this chapter.

4.2 PMP quantization, PDP modeling and Thermal Equation

As discussed in the previous chapter, the Thread Interference Power (TIP) and Thread Solitary Power (TSP) model deduces the additional power dissipates through the multiprocessing of application threads. The TIP-TSP computation approach in the previous chapter relies on addition system power monitoring module. In order to build a standalone Linux Scheduler, without determining offline system power monitoring facilities, a TIP-TSP monitoring and computation approach using existing Linux modules and on chip sensors is proposed in this section.

The Linux commands `perf` and `oprofile` are utilized to record detailed Performance Metric Parameters (PMPs) such as cache miss and hit rates, Translation Lookaside Buffer (TLB) hit and miss rate, and number of context switches. The relevant PMP readings are generated by counting number of events with on chip Performance Monitoring Units (PMU).

The proposed Scheduler, named as the Re Scheduler aims to use the individual PMP counts to construct the TIP-TSP model at runtime and making scheduling decisions solely based on the amount of PMP saved without measuring the power dissipated directly. However the number of PMP events is not directly translatable to system performance costs, as the number of CPU cycles involved in a miss event of different memory unit such as the L1 cache and the L2 cache are not equal and can be orders of magnitude more or less.

Hence it requires normalizing PMPs according to their cost in number of CPU cycles prior to the computation of TIP-TSP. The normalization is done by quantifying the cost of each PMP and translating the PMP counts into the number of cycles according to the corresponding cost. The PMP quantization approach and the normalizing techniques are discussed below.

Other than discussing the computation of TIP-TSP at runtime, a novel approach of separating the TIP and TSP from the total power dissipated is also discussed in this section. As shown in Table 4.1, there are 25 PMPs considered in the computation of the TIP-TSP. It will be rather computational intensive if each individual PMP is considered at each sampling interval. Thus a group of Power Dissipation Problems (PDPs) are identified to constrain the number of PMPs considered while separating the TIP and the TSP from the total CPU power consumed.

Table 4.1 Performance Metric Parameters

L1 dCache Metric		L1 iCache Metric		L2 Cache Metric		dTLB Metric	
1	L1-dcache-loads	7	L1-icache-loads	11	LLC-loads	16	dTLB-loads
2	L1-dcache-load-misses	8	L1-icache-load-misses	12	LLC-load-misses	17	dTLB-load-misses
3	L1-dcache-stores	9	L1-icache-prefetches	13	LLC-stores	18	dTLB-stores
4	L1-dcache-store-misses	10	L1-icache-prefetch-misses	14	LLC-store-misses	19	dTLB-store-misses
5	L1-dcache-prefetches			15	LLC-prefetch-misses	20	dTLB-prefetches
6	L1-dcache-prefetch-misses					21	dTLB-prefetch-misses
iTLB Metric		Branch Metric					
22	iTLB-loads	24	branch-loads				
23	iTLB-load-misses	25	branch-load-misses				

Each PDP denotes a pattern of addition power dissipates through TIP that exerts in only a certain number of PMPs, and the rest of the PMPs has no or little correlation with the resultant TIP. Hence upon the determination of PDP, only the relevant PMPs are considered at the TIP and TSP separation, which inflicts less computation overhead as comparing to consider all PMPs.

Figure 4.1 shows the process of PMP normalization and followed by a TIP-TSP decomposition based on the PMP selected with PDP identified. Where the TSP and the TIP terms can either share the same PMP or consist of a different set of PMP based on the structure of the PDP selected.

Figure 4.1 PMP normalization and PDP based TIP-TSP decomposition

4.2.1 Performance Metric Quantization

In this section the cost of each Performance Metric Parameter is quantified as numbers of CPU cycles. The impact of task migration and context switches are also calculated.

Cache Miss

Caches are used to reduce the access time for the CPU to retrieve working sets from the main memory. The number of CPU cycles involved can be reduced up to 91% by predicting and caching working sets needed for process execution in the cache. The cache entries are stored as data lines instead of data words. Number of

transfers needed to load up a cache line is computed using the cache line length/the memory bandwidth. In Cortex A series processors the L1 cache line size is 64 bytes and the memory bandwidth of a DDR3 RAM is 64 bits wide.

To gauge the cost of a cache miss, the access time of different memory elements is studied. The time taken to access a register $Time_{register\ access}()$ in the CPU is denoted as a unit cycle of 1. The time taken to read a cache line of 64 byte in L1 cache is represented as,

$$Time_{L1\ cache\ load\ hit} = \frac{Size_{L1\ cache\ line}}{Size_{movap}} \quad (4.1)$$

Equation (4.1) is the load hit time, where $Size_{L1\ cache\ line}$ is the size of L1 cache line in bytes and $Size_{movap}$ is the number of bytes the `movap` instruction is capable of moving in 1 cycle. It is computed as the time taken for the `movap` instruction to move 16 bytes a cycle, and the result is 4 cycles. Once there is a cache miss in the L1 cache, L2 cache entries will be walked to find the requested memory address. Hence the L1 cache miss time is equivalent to the L2 cache hit time $Time_{L1\ cache\ miss} = Time_{L2\ cache\ hit}$. Upon allocating the requested memory address in L2 cache, the read interface from L2 to L1 cache will be utilized to read the relevant data in the L2 cache, in Cortex A series processors the read interface is 128 bits wide. Hence the time taken to access L2 cache is calculated with Equation (4.2)

$$Time_{L2\ cache\ hit}(s) = Time_{L1\ cache\ miss\ resolve} + Time_{L2\ cache\ walk}(s) + Time_{L2\ to\ L1\ cache\ load} \quad (4.2)$$

where $Time_{L1\ cache\ miss\ resolve} = 3$ is time taken to resolve the L1 cache miss; $Time_{L2\ cache\ walk}(s) = 3$ is the time taken for the cache logic to walk the entire L2 cache for the given cache tag, the ideal value is 3; s denotes the size of an element in a Working Set (WS), which affects prefetching mechanism and the time taken to walk the L2 cache; $Time_{L2\ to\ L1\ cache\ load} = \frac{Size_{L1\ cache\ line}}{Size_{L2\ to\ L1\ read\ interface}}$ is the time taken to load the L1 cache line with the L2 cache data. The time taken is bounded by the bandwidth of the read interface between L2 and L1 cache $Size_{L2\ to\ L1\ read\ interface}$ as the L1 cache line size $Size_{L1\ cache\ line}$ in a 64 bit system is normally 64 byte. In an ideal case the targeting cache line in L2 cache will be prefetched if the element size s is smaller than 2 times of the cache line and the term $Time_{L2\ cache\ walk}(s)$ can be eliminated.

If the L1 cache miss cannot be resolved with the entries in the L2 cache or the Last Level cache, the main memory will be accessed to look for the request memory address. The memory access time differs according to the distance between the processing CPU core and the DRAM location. The average access time to local DRAM is

$$Time_{DRAM\ local} = Time_{L2\ cache\ miss}(Local) = 200\ cycles \quad (4.3)$$

and the access time to remote DRAM is

$$Time_{DRAM\ remote} = Time_{L2\ cache\ miss}(Remote) = 300\ cycles \quad (4.4)$$

Figure 4.2 shows the time taken to access different memory units, the block on the left indicates the instructions waiting in the run queue.

Figure 4.2 Memory Unit access latencies

The cache hit and miss latencies of various caches change according to the architecture and setup of the system. Hence an initialization script is executed to quantize the cache Performance Metric Parameters. The script creates a Working Set with Working Set Size (WSS) = 2^n bytes and fills the Working Set with a variable `E` with a data type of `struct` with size of the `struct` = $64 \times s$ bits. The multiple `structs` created are linked to each other as a linked list. The Working Set is then stored in the memory with the `structs` laid out sequentially. The script will then walk through the list and the average time taken to read one `struct` is recorded to denote the base access time of various caches. The walk through will repeat for different WSS. Figure 4.3 shows the result of the script running in a CPU with 64 KB of L1 cache and 2 MB of L2 cache. The size of the `struct` = 128 byte which is 2 times the size of the cache line and the purpose is to neutralize the function of the prefetch mechanism in order to get the accurate latency of accessing different memory elements. As shown in Figure 4.3 (where n indicates the size of the WSS in 2^n bytes) the average L1 cache hit time is 4 cycles, from WSS = 2^{10} to WSS = 2^{16} . From WSS = 2^{17} onwards the number of cycles taken to access the `struct` is dominated by the L2 cache access time, which is because of the elements in the Working Set are sequentially linked. In a sequentially linked Working Set the request to access the part of the Working Set that is not in the L1 cache will generate a L1 cache miss and walk through the L2 cache for the memory address. The steep edge at WSS = 2^{20} for the same reason and the DRAM access time dominates the time taken to access the Working Set.

Figure 4.3 Cache hit time of a sequential Working Set

Figure 4.4 Cache Miss Rate of a random Working Set

In random access scenarios, the cache hit and miss ratio increases according to the ratio between the WSS and the cache size. The cache miss rate of the L1 and L2 cache can go up to 25% for processes with WSS that exceeds the size of the cache. Figure 4.4 shows a scenario of the initialization script generates Working Set with random ordered elements instead of sequential. The average Effective Access Time (EAT) of a memory unit can be expressed with the influence of the Miss Rates with the following equation,

$$EAT = Time_{memory\ unit\ hit} \times HitRate_{memory\ unit}(\omega) + Time_{memory\ unit\ miss} \times MissRate_{memory\ unit}(\omega) \quad (4.5)$$

where $Time_{memory\ unit\ hit}$ denotes the ideal number of cycles taken for a memory unit access and the memory unit can be L1 or L2 cache; $HitRate_{memory\ unit}(\omega)$ denotes the hit rate of the memory unit under WSS ω ; $Time_{memory\ unit\ miss}$ denotes the access time taken when the memory unit is missed; $MissRate_{memory\ unit}(\omega)$ denotes the miss rate of the memory unit under WSS ω . Since the Miss Rate is positively related to the Working Set Size, the initialization script constructs the relationship between the WSS and the Miss Rate of each memory unit. The relationship is used to predict the amount of additional memory unit miss events while working on threads with large WSS.

In the actual Scheduling Scheme application, the HOLD scheme can be utilized to reduce the miss rate by limiting the WSS by grouping threads with small WSS to execute simultaneously, scheduling threads with large WSS to execute alone as much as possible.

TLB miss

The Translation Look-Aside Buffer (TLB) is a small cache in the Memory Management Unit (MMU) used to store the recent translation from virtual memory address to physical address mapping for fast retrieval upon referencing. Similar to the L1 and L2 cache, the TLB prefetches a number of translated virtual to physical memory address mappings prior to the retrieval request according to the data accessing patterns. There are normally multiple levels of TLB in a Memory Management Unit. For example in the Cortex A architecture

there is a micro TLB with 10 entries residing in both the L1 instruction and data cache, and a main TLB with 512 entries [108]. The hit time of TLB can be very small as each entry is 32 bits long in 32 bits mode and can be up to 40 bits long in the 64 bits mode. Hence the TLB hit time is much lower compared to the L1 cache hit time,

$$Time_{TLB\ hit} = \frac{Size_{TLB\ entry}}{Size_{movap}} \quad (4.5)$$

where $Size_{TLB\ entry}$ denotes the size of a TLB entry, $Size_{movap}$ is the number of bytes the `movap` instruction is capable of moving in 1 cycle, $Time_{TLB\ hit}$ denotes the TLB hit time which is less than a cycle and is approximated to be 1 cycle. A TLB miss happens when the virtual memory address requested does not match an entry in the TLB, and an expensive page table walk will be conducted in order to locate the requested memory address. The cost of a TLB miss is bounded by two factors, the type of the memory unit that the page table entry indicating cache line is stored and the number of times that table has to be walked to find the cache line. The number of walks is bounded by the size of the page size, more walks are required as the size gets smaller [109]. The TLB miss time is expressed with Equation (4.6).

$$Time_{TLB\ miss}(\rho) = Time_{memory\ unit\ hit} \times (Trip_{page\ translation\ retrieval}(\rho) + 1) \quad (4.6)$$

where $Time_{TLB\ miss}$ is the time taken to resolve a TLB miss; $Time_{memory\ unit\ hit}$ denotes the time take to access the page table. It differs according to the type of memory unit used to store the page table, as generally the page table is stored in the main memory. Hence the average DRAM access time of 250 cycles is used; $Trip_{page\ translation\ retrieval}(\rho)$ denotes the number of trips needed to resolve the page translation, and the number varies according to the page table size ρ . There is an additional last trip to finally fetch the data. Hence in total there are 3 trips to the memory to load the data mapped on a 4KB page, 2 trips for a 2MB page and 1 trips for a 1GB page. Each trip costs as much as 250 cycles.

To accurate deduce the average time taken to resolve a TLB miss, the Performance Monitoring Unit samples in the result of random Working Set from the initialization script is analyzed. The PMU `PM_DATA_TABLEWALK_CYC` is used to denote the total number of table walk cycles resulted from TLB miss and `PM_DTLB_MISS` to denote the number of TLB miss counts [110]. Only dTLB (data TLB) is involved in the computation as iTLB (instruction TLB) is less messy and generate much less overheads. The WSS is set to be 2^{22} bytes and the struct size to 128 byte to overflow the TLB and disable the prefetch as much as possible. With the example TLB miss and Page walk cycles shown in Figure 4.5, the resulted TLB miss time calculated with $\frac{\text{Total Page Walk Cycles}}{\text{Total Number of TLB miss}}$ is 630 cycles, which is reasonable and within the range of $Time_{TLB\ miss}(\rho)$.

Figure 4.5 Number of TLB miss and Page walk cycles

Figure 4.6 shows the relationship between the WSS and the TLB miss rate, where the larger Working Set Size results in higher Effective TLB miss rate as referencing to Equation. (4.5). Hence the variation in WSS is also taken into account while computing the resulting TIP. The essence of keep TLB miss rate and the cost of a table walk low is to reduce the number of directory entries in the TLB by increasing the size of a page. Hence using HOLD scheme to run processes and allocating their pages onto a larger page can reduce the number of TLB miss. The DEDICATE scheduling scheme and SUPER scheduling scheme can be applied together to reduce the number of context switches that causes excessive TLB misses and the details are discussed in Section 4.5.

Figure 4.6 TLB Miss Rate of different WSS

Branch miss predictions

The mission of branch prediction is to predict which branch to go to before the conditions are certain, this hides the waiting time while resolving a conditional jump. In a branch prediction hit event there is no additional access time added to the CPU cycle. In the event of a branch prediction miss event the CPU instruction pipeline gets flushed and the numerous CPU cycles of effort in resolving the branch are reverted back to the branch point of the nearest converge point [45].

Figure 4.7 shows a branch created by a conditional jump. Upon a branch misprediction the code blocks in the mispredicted path gets executed. In the worst case, the branch misprediction does not get resolved early at the decoding and executing stage, and only gets detected at the Reorder Buffer (ROB) after executing a few

code blocks. Because of the divergence of the branch whatever has been previously executed in the mispredicted path is discarded and the correct path will be executed from the branch point. Figure 4.8 shows a branch created by an if-else condition, similar to the example shown in Figure 4.7. Upon a branch misprediction the mispredicted path gets executed until a worst case branch misprediction detection in ROB. However this branch converges at the end of the coding block. In such case it does not have to revert all the way back to the branch point. The branch misprediction can be resolved with executing the correct path from the convergence point. However the CPU instruction pipeline gets flushed upon a branch misprediction regardless of the convergence of the branches.

Figure 4.7 Branch Resolution in Divergence

Figure 4.8 Branch Resolution in Convergence

As shown in Equation (4.7), the cost of a branch prediction (BP) miss $Time_{Branch\ miss}$ consists of two parts: the CPU refill time $Time_{Pipeline\ refill}$ which is the same as the length of the CPU pipeline. For example the Cortex A CPU pipeline is 20 cycles in length; and the branch resolution time $Time_{Branch\ resolve}(C)$ which is the time taken to detect the branch misprediction and carry on executing the code blocks in the correct path. The branch resolution time is bounded by the convergence of the branch C . When the branch is absolute divergent the time taken to resolve a branch miss is equivalent to the CPU refill time, in the case of the branch being absolute convergent the time taken to resolve the branch miss is nearly negligible. In the computation of the TIP and quantization of the PMP, C is given as being absolute convergence at 50% of the time and 50% of absolute divergence as suggested in [46], which gives an overall 30 cycles of branch prediction miss time in Cortex A architecture.

$$Time_{Branch\ miss}(C) = Time_{Pipeline\ refill} + Time_{Branch\ resolve}(C) \quad (4.7)$$

There are a lot of different factors affecting the miss rate of a branch prediction, such as different types of workload. To determine the baseline of the branch prediction miss rate SPEC CPU 2006 benchmark has been used, BP miss rate of different benchmarks has been generated and recorded, the average values is used as the baseline BP miss rate. As shown in Figure 4.9, the average branch prediction miss rate is 13%. *AppSets* that increase the BP miss rate over this value are considered as generating excessive branch misprediction.

Figure 4.9 Branch Prediction Miss Rate of different Benchmarks

Context Switching

By default in Linux a running process operating under non-real-time scheduling policy will be given a time slice with a minimum length defined by the minimum granularity `sysctl_sched_min_granularity = 750ns` (1500 cycles in a 2 GHz processor) and the maximum length constrained by the latency `sysctl_sched_latency = 6000ns` (12000 cycles in a 2 GHz processor). The CPU time or the Quantum allocated to a process operating under the real time policy is by default 100,000ns or 200,000 cycles.

An initialization test has been conducted to calculate the time cost of a context switch. Two parallel executing threads are deployed and the `futex` system call is used to make the two threads waiting for each other in turn. At each iteration the time between the executions of the two threads are computed and after 500 iterations the average time taken is noted as the context switching time. Figure 4.10 shows the result of the initialization script executed on a Cortex A processor with 2 GHz of operating frequency. In the example a typical context switch across different CPU cores can cost up to 10000 cycles or an average 8000 cycles and a context switch within the CPU core costs an average of 4000 cycles. The figure may vary according to different CPU architectures. Hence the initialization script is executed upon scheduler deployment to compute the baseline context switching cost of the targeted platform.

Figure 4.10 Process of a context switch

The context switching cost discussed above is named as the direct CPU cost. There is the indirect context switching cost that has been systematically studied by a team from University of Rochester in [50]. Upon a successful context switch, the amount of misses in L1 cache and consecutive memory units will increase momentarily. That is due to changing in processes cost a mismatch between the working set requested by the thread and the working set stored in the memory units. Equation (4.8) shows cost of a context switch measured in time,

$$Time_{Context\ Switch}(t, P, \omega) = Time_{Context\ Switch\ Direct}(t) + Time_{Context\ Switch\ Indirect}(t, P, \omega) \quad (4.8)$$

where, $Time_{Context\ Switch}(t, P, \omega)$ is the time taken of a context switch, which is bounded by the working set affinity P , type of switch t and the working set size ω ; $Time_{Context\ Switch\ Direct}(t)$ is the direct cost inflicted by a context switch, as calculated in the example shown in Figure 4.10; t denotes whether the switch is within the core or to another core; $Time_{Context\ Switch\ Indirect}(t, P, \omega)$ is the indirect context switch cost, where ω the effect of the Working Set Size on the context switching time and P denotes if the two switching threads are from the same process.

Initialization test has been conducted to compute the indirect context switching cost of the targeting platform, which is Cortex A in the example shown in Figures 4.11 and 4.12. Context switch between threads from the same process and threads from different processes has been under the context switching scenario of switching within the core and across different cores. Figure 4.11 shows the result of $t = 0$, which denotes that the switch is within the core. The blue line denotes the context switch time between the threads from the same process $P = 0$ and the red line denotes the context switch time between threads from different processes. From the result it is observed that the average indirect context switching cost across different working set sizes is $Time_{Context\ Switch\ Indirect}(t = 0, P = 0) = 1120$ cycles and $Time_{Context\ Switch\ Indirect}(t = 0, P = 1) = 1660$ cycles. The difference between the indirect context switch costs is due to the additional cache miss rates resulted from cache interference between different work sets. In Linux during the context switching of two threads from the same process, the register CR3 is not updated as there is no memory structure difference between the switching threads as what is commonly found in inter process switching. The content of CR3 denotes the pointer to the page table, while not updating it is equivalent to not changing the page table and hence does not cause a flush to the TLB. Thus there is less cache interference between the thread from the same process.

Figure 4.11 Context switching cost within the CPU core

Figure 4.12 shows the impact of the indirect context switching cost when $t = 1$ or switching between different CPU cores. The inter core indirect context switching cost can easily cost 20 or 30 times more than the intra core switch, which demonstrates the migration of cache content from one core to another. The process involves a vast amount of CPU time to go through the hierarchy of different memory units. Thus it is important to avoid inter core context switch. It shows the advantage of setting the CPU affinity to the thread, which can constraint the context switching to take place within the same CPU core.

Figure 4.12 Context switching cost across CPU cores

Similar to the memory units addressed earlier in this section, the Working Set Size increases the context switching time and the indirect cost is a result of moving Working Set across memory units. When WSS at (2^{16} bytes = 64KB) is larger than the L1 cache (64KB), the switching cost is bounded by the TLB miss rate and the turnaround time to walk a memory page. A memory page walk costs 10 times more CPU cycles comparing to accessing caches locally. Thus, there is a surge in context switching cost at $WSS = 2^{16}$. It is better to keep the Working Set Size in mind while computing the context switching cost.

In the actual scheduling using the proposed Scheduler, SUPER scheme has promised to reduce the overall number of context switches by allocating larger time slices and the DEDICATE scheme reduces inter core context switch by binding the threads with CPU cores.

Memory Impact

Upon context switch, the entire TLB is flushed, which is 128 entries in the Cortex A context. One entry is 128 bits, hence it takes 256 cycles to rewrite the TLB.

Setting CPU affinities for newly created process thread can reduce the amount of context switches, cache miss rate and memory access time in a NUMA environment. The `set_mempolicy` is used to set memory policy for a given thread, where the memory allocated to the targeting thread will be based on a given set of nodes. Hence in the DEDICATED scheme the NUMA nodes are set together with the CPU affinity.

The kernel has the authority to allocate memory of the targeting thread onto other nodes if the memory around the preferred nodes are running out. Hence it is very likely that the different multiplexing threads

allocated to a CPU core may ended up have difference memory layout if all are set with local node memory locating policy. Therefore the NUMA affinity is also considered as a part of the TIP.

Just like migrating threads between CPUs for better affinity, memory pages can be moved to the nodes nearer to the CPU core that is executing the targeting thread using the `MPOL_MF_MOVE`. However the resulting interference that may impact the memory allocation of other threads should be considered as a cost or a penalty term to balance the memory access gain from the move.

The CPU to task mapping based on CPU affinity and memory node NUMA distance can be set with the `cpuset` and `nodeset` in Linux kernel. Utilizing NUMA aware memory allocation policies has shown a 10-15% performance increase [111] and an average 50% of memory access time reduction as comparing to fully random allocated memory [112].

4.2.2 Thermal Hotspot and Power Dissipation Quantization

Power

The total power consumed by CPU consists of two components [113], the static power and the dynamic power $P_{CPU} = P_{static} + P_{dynamic}$. The dynamic power $P_{dynamic}(V, f) = C \times V^2 \times f$ increases as the supply voltage and operating frequency increases. The default voltage frequency pair of a CPU is named as the Operating Performance Points (OPP). In the context of CPU power management, Dynamic Voltage Frequency Scaling (DVFS) enables the system to choose amongst varies OPPs for optimized performance and power efficiency. The static power $P_{static}(V, Tmp) = \beta \times V^2 \times e^{-\frac{E-D\sqrt{V}}{273.15+Tmp}}$ describes the power lost by the current traveling from the source to the ground. The term static gives a good characterization of the power loss, it is dissipated as long as the power is on regardless of the switching activities of the logic gates. The static power is highly temperature sensitive like any other semiconductors that suffer the temperature effect. Thus as in the equation the static power is bounded by two variables voltage V and temperature Tmp . β, E and D are the coefficients that vary between different architectures and requires to be determined non parametrically. $e^{-\frac{E-D\sqrt{V}}{273.15+Tmp}}$ is derived from the Frenkel-Poole emission [114] that is used to model the leakage current variation with reference to the of density of traps in the semiconductor. The original equation is modified to model the effect of temperature on the leakage current. The overall CPU power consumption is then modeled with the following equation,

$$P_{CPU}(V, Tmp, f) = \beta \times V^2 \times e^{-\frac{E-D\sqrt{V}}{273.15+Tmp}} + C \times V^2 \times f \quad (4.9)$$

To determine the coefficient non-parametrically, initialization script is executed with different CPU voltage and frequency pair OPP acquired with Linux OPP library. To constrain the test to a single core in a multi-core architecture CPU Hotplug is performed in the Cortex A57 example. The CPU core temperature is sampled every second with `lm-sensor` and the instantaneous power dissipation is sampled with `power_now` initialization script. The collected reading are then used to fit Equation (4.9) to determine the unknown

coefficients. The initialization script runs the test by set when the base temperature is detected, during the running of the set the voltage value is increased to reach individual OPP of the CPU core. The power value is sampled every second and the test continues till the CPU temperature drops to the next available based temperature. Figure 4.13 shows the initialization test executed to determine P_{CPU} coefficients of a Cortex A57 CPU core across 8 base temperatures. The vertical axis (y axis) shows P_{CPU} values collected in watts, the primary horizontal axis (x axis) shows V measured in volts and the secondary horizontal axis (z axis) shows T in degree Celsius. As operating frequency f varies linearly with the voltage V , x axis indicates only V for a clearer presentation. Each point on the surface represents the amount of CPU power dissipated P_{CPU} at a specific OPP (V, f) and temperature T . By substituting the values shown in Figure 4.13 into Equation (4.9), a system of polynomial equations will be generated. For example, by substituting P_{CPU}, V, T, f values at the points $\{P_{CPU}(1.037, 30, 2.1) = 4.23W, P_{CPU}(0.99, 40, 2) = 4.04W, \dots, P_{CPU}(0.92, 55, 1.9) = 3.64W\}$, a system of polynomial equations is generated as $\{4.23 = \beta \times 1.037^2 \times e^{\frac{E-D\sqrt{1.037}}{273.15+30}} + C \times 1.037^2 \times 2.1, 4.04 = \beta \times 0.99^2 \times e^{\frac{E-D\sqrt{0.99}}{273.15+40}} + C \times 1.037^2 \times 2, \dots, 3.64 = \beta \times 0.92^2 \times e^{\frac{E-D\sqrt{1.037}}{273.15+55}} + C \times 0.92^2 \times 1.9\}$. The system of polynomial equations is then solved simultaneously with Matlab numeric solver for solutions of $\{\beta, E, D, C\}$. The solved coefficient values are $\beta = 195.4, E = 2013.3, D = 589.7, C = 0.9$.

Figure 4.13 Cortex A57 CPU core power dissipation at different OPPs and temperature

Thermal

As in Equation (4.9), power dissipated in CPU will cause its temperature to rise. Hence the impact of power dissipation and the CPU temperature has to be investigated prior to accurately monitor the relationship between the performance and the power dissipated. The lumped RC temperature model from [115] has been adopted, assuming that the targeting temperature is modeled as a continuous time series. The temperature sampled during the period from any given point in time t with a sampling interval length of n consists of two components, the instant state temperature and the transient state temperature $Tmp(t + n) = Tmp_{inst} - T_{trans}$. The instant state temperature $Tmp_{inst} = P_{CPU} \times R + Tmp_{room}$ describes the amount of temperature generated by the total CPU power consumption at the given sampling interval; R denotes the thermal resistance in $\frac{^{\circ}\text{C}}{\text{W}}$, which is architecture dependent and needs to be derived as a coefficient with nonparametric method; Tmp_{room} denotes the room temperature, which is 30°C as the mobile devices are normally kept near to human body. However with T_{inst} alone it is not able to model the thermal energy loss through ambient and heat sink. Hence the resulted temperature is complimented with the transient term. The transient state temperature $Tmp_{trans} = (P_{CPU} \times R + Tmp_{room} - Tmp(t))e^{-\frac{n}{RC_t}}$ describes the thermal energy gain and loss with reference to the sampling interval length n and temperature at the beginning of the sampling $Tmp(t)$. The transient state computes the difference between the instance state temperature from t to n and the temperature at time t . If the sampling interval is extremely short the overall temperature $Tmp(t + n)$ will be dominated by $Tmp(t)$, and in the case of a very long sampling interval $Tmp(t + n) = Tmp_{inst}$ as the $Tmp_{trans} \approx 0$ with very large n . C_t is the thermal capacitance $\frac{\text{Joules}}{^{\circ}\text{C}}$ that is treated as a coefficient and determined non parametrically. The temperature at time t with a sampling interval of n is then given as Equation (4.10).

$$Tmp(t + n) = P_{CPU} \times R + Tmp_{room} - (P_{CPU} \times R + Tmp_{room} - Tmp(t))e^{-\frac{n}{RC_t}} \quad (4.10)$$

To determine the coefficient in Equation (4.10) non-parametrically, initialization script is executed in the similar manner as used to determine the coefficient in Equation (4.9). The temperature reading acquired with digital thermal sensors is sampled every n seconds interval, which is 1 second by default. The gradient between Tmp_t and Tmp_{t+n} is computed. When the gradient is less than the threshold gradient $\Delta Tmp_{th} = 1\%$, it denotes that the temperature is dominated by the T_{inst} . Then the value of the thermal resistance R is computed with the T_{inst} equation and the power reading at $t + n$. A sampling interval length n that can lead to a less than threshold gradient is used to determine the thermal capacitance coefficient C_t . The test is executed against different OPP switching ranges, from a base OPP with relatively lower frequency to a peak OPP with a higher operating frequency.

Figure 4.14 shows the initialization test executed to determine $Tmp(t + n)$ coefficients of a Cortex A57 CPU core, the base OPP frequency is 1.2 GHz, while the Peak frequencies are from 1.3 GHz to 2.1 GHz. The values of $\{R, C_t\}$ are solved using the same method as addressing in solving coefficients for Equation (4.9). The solved coefficient values are $R = 6.87 \frac{^{\circ}\text{C}}{\text{W}}$, $C_t = 14.7 \frac{\text{Joules}}{^{\circ}\text{C}}$.

Figure 4.14 CPU core Temperature Change switching from 1.2 GHz to different OPP

Energy Budget Calculation

While defining the power dissipated in the CPU cores at a given voltage and frequency, the power density of different CPU function blocks is not much a concern in the real time scheduling environment, because there is not an efficient, adequate and accurate mean to monitor and gauge the power variation on a per core per function block basis. Hence the power density of a single CPU core is considered as uniform as in Equation (4.9). Similar assumption has been adopted to validate the CPU core temperature. The peak temperature detected by the temperature sensor in each CPU core is taken as the average temperature of the core, which is indeed not exactly modeling how the temperature varies across the function blocks of the core, but it is able to provide a worst case temperature budget for operating frequency determination in the proposed scheduler.

Whenever there are cycles saved from remedying TIP dissipation, the frequency, temperature and voltage of the CPU are recorded together as the power saved $P_{saved}(V, Tmp, f)$. The processor time saved is the same as the sampling period used in Equation (4.10). It is denoted as $n = \frac{Cycles_{saved}}{f}$, which is computed with the number of cycles saved and the operating frequency of the CPU during the sampling period. Energy saved $E_{saved}(V, Tmp, f, n) = P_{saved}(V, Tmp, f) \times n$ is calculated as the product of the power saved and the sampling period, and the summation of the energy saved at varies TIP resolutions constructs the energy budget E_{budget} . The energy budget denotes the amount of energy that is allocable when the Re Scheduler is making a decision to increase the OPP by making use of the `CPUfreq` governor [69] of Linux. Increasing OPP results in increased voltage and frequency, and according to Equation (4.9) there will be additional

dynamic power dissipated as the voltage and frequency increase. The static power dissipation from the temperature raised due to the increased voltage and frequency is as described in Equation (4.10).

Prior to increasing the OPP, the sampling length of a given frequency is computed with following steps: 1. Calculate the amount of $P_{CPU}(T_{mp_t}, f, V)$ that is going to be generated by the chosen frequency and voltage pair of the OPP at current temperature; 2. Compute the amount of temperature raise due to the increasing P_{CPU} using Equation (4.10). Figure 4.15 shows an example of CPU temperature fluctuation VS time as the operating frequency increased from 1.2 GHz to 2.5 GHz at time 40 seconds and switches back at 70 seconds; 3. Calculate the additional power dissipated due to increased temperature and OPP using Equation (4.9).

Figure 4.15 Temperature rise from 1.2-2.1 GHz

Figure 4.16 shows an example of additional power dissipated due to OPP switching. Computing the shaded area gives the amount of additional energy dissipated E_{diss} ; 4. Compare additional energy dissipated with the energy budget, and adjust sampling interval n to satisfy the condition $E_{diss} < E_{budget}$.

Figure 4.16 Power dissipation rise from 1.2-2.1 GHz

The OPP and sampling interval computation steps are used to accurately derive the impact of the DVFS operations applied by the Re Scheduler.

4.2.3 Power Dissipation Problem Categorization

As addressed in Chapter 3, the apps or applications which are executing concurrently form an $AppSet(seq)$, where seq denotes the entry sequence number of the $AppSet$ in the $AppSet Table$. Relevant information such as the Thread Interference Power (TIP) and Thread Solitary Power (TSP) are stored under the $AppSet Table$ entry of the targeting $AppSet$.

The total power dissipated by the application threads in the $AppSet$ consists of two components, $P_t(AppSet(seq)) = P_s + P_d$, where P_s is the Thread Solitary Power (TSP) and P_d is the Thread Interference Power (TIP). Based on what has been discussed in Chapter 3, the Re Scheduler follows the same TIP-TSP modeling approach to remedy the additional power dissipated through TIP.

Different application threads have different system resource requirements and execution patterns at runtime. As a result the interference caused by an application thread depends on the other threads with which it is executing concurrently. Hence the amount of TIP contributed by any individual thread varies across different $AppSets$. The TIP and TSP dissipate through several stages of thread execution: executing instructions in pipeline; storing/loading in cache; looking up memory address in TLB; context switching activities. The power dissipation is monitored and computed by normalizing the number of counts of the 25 Performance Metric Parameters shown in Table 4.1. As the TIP only occurs upon conflicts of execution resources between threads in an $AppSet$ and such conflicts do not occur in all the execution stages of application threads, not all PMPs will show additional number of counts to deduce the TIP using $P_d = P_t - P_s$.

The different conflicts that generate TIPs in different PMP patterns are named as the Power Dissipation Problems (PDPs), which are used to reduce the amount of PMPs used to calculate TIP and complexity to update the $AppSet Table$ at runtime. The total power dissipated by an $AppSet$ is represented in the PMP format using $PMP_{total}(AppSet) = PMP_{TIP}(AppSet) + PMP_{TSP}(AppSet)$. Each Power Dissipation Problem is modeled with a masking vector to eliminate the trivial PMP terms in the calculated TIP in PMP format $PMP_{TIP}(AppSet)$. A PDP masking vector is denoted as $PDP(q) = \{\tau_1, \tau_2, \dots, \tau_i\}$, where q denotes the sequence number of the PDP, $\tau_i = [0,1]$ denotes the threshold value for the given the corresponding PMP value ρ_t^i from $PMP_{total}(AppSet) = \{\rho_t^1, \rho_t^2, \dots, \rho_t^i\}$ position i ; The Threshold value τ_i is deduced by $\frac{\rho_{TIP}^i}{\rho_{TSP}^i}$, hence the value of τ_i equals to zero if there is no TIP dissipated and equals to a fraction between 0 to 1 if there is TIP dissipated.

Computation of TIP using the PDP mask adopts different procedures depends on the $AppSet$ as described below:

1. If the $AppSet$ is previously unseen by the Scheduler and creates a new entry in the $AppSet Table$, the $PMP(AppSet_{new})$ is then masked with all PDP masking vectors used to deduce PDP ranking using Equation (4.11).

$$PDP_{rank}(q) = \sum_{i=1}^{25} PDP(q) \times \left[\frac{PMP_{TSP}(AppSet_{new})}{PMP_{total}(AppSet_{new}) - PMP_{TSP}(AppSet_{new})} \right] \quad (4.11)$$

where, $PDP_{rank}(q)$ denotes the rank of the Power Dissipation Problem q . The PDP with the highest rank is denoted as the PDP of the $AppSet$ and recorded in the $AppSet$ Table; PMP_{TSP} denotes the TSP in PMP format and is computed by walking the $AppSet$ Table and summing the TSP values of all member threads of the given $AppSet$; the divisor term $PMP_{total}(AppSet_{new}) - PMP_{TSP}(AppSet_{new})$ computes for the TIP of $AppSet_{new}$ in PMP format $PMP_{TIP}(AppSet_{new})$, which is equivalent to compute the Hadamard product of $PMP_{TSP}(AppSet_{new}) \odot PMP_{TIP}(AppSet_{new})^{-1} = \left\{ \frac{\rho_{TSP}^1}{\rho_{TIP}^1}, \frac{\rho_{TSP}^2}{\rho_{TIP}^2}, \dots, \frac{\rho_{TSP}^i}{\rho_{TIP}^i} \right\} = \left\{ \frac{1}{\tau_1}, \frac{1}{\tau_2}, \dots, \frac{1}{\tau_i} \right\}$, where $\odot$ is the element wise vector product notation. Thus the rank of $PDP(q)$ is computed as a group of 0s and 1s; in the cases where TIP dissipated is more than the threshold then there is a possibility that the $AppSet$ does not get best described by the best fitted PDP but the PDP contributes the largest rank values according to the excessive TIP dissipation. At the end of the computation the PDP vector $PDP(AppSet_{new}) = \max_{1 \leq q \leq 9} PDP_{rank}(q)$ and the TIP of the $AppSet_{new}$ are updated in the $AppSet$ Table.

2. If the $AppSet$ is old as it is previously monitored and recorded in the $AppSet$ Table, its TIP is computed with Equation (4.12):

$$\begin{aligned} PMP_{TIP}(AppSet_{old}) &= Mask(AppSet_{old}) \odot PMP_{total}(AppSet_{old}) \\ &\quad - Mask(AppSet_{old}) \odot PMP_{TSP}(AppSet_{old}) \end{aligned} \quad (4.12)$$

where, $Mask(AppSet_{old})$ is the masking vector for $AppSet_{old}$ that consists of 0s and 1s. It is the Hadamard product of the PDP mask vector and the inversed vector of $PDP(AppSet_{old}) \odot PDP(AppSet_{old})^{-1}$. The masking is applied to the PMP_{total} and PMP_{TSP} during the PMP monitoring phase.

To determine the corresponding threshold parameters in each PDP masking vector, the characteristic of the TIP dissipation patterns has been studied. SPEC CPU 2006 benchmark was used to prepare the test sets. Each test set contains a random composition of benchmarks executing in parallel. Then benchmarks results are grouped according to the PMP patterns in their test results and each group denotes one Power Dissipation Problem. The result of executing the selected benchmarks individually denotes the TSP of each test, and the TIP is computed using the PMP_{total} equation.

Figure 4.17 shows the computed PDP threshold of 9 different Power Dissipation Problems. Figure 4.17 demonstrates the PDP threshold distribution for a given Cortex system, which varies from system to system. The vertical axis indicates the threshold value measured for each PMP indicated in the primary horizontal axis, and the secondary axis indicates threshold values from τ_1 to τ_{25} for each given PMP as stated in the figure. Given a set of monitored PMP readings, PDP s can be determined by comparing the monitored values against the threshold values. A PDP is marked as present, if the PMP readings exceed its corresponding PMP threshold computed in the figure. The PMP threshold readings of each PDP are summarized in Table 4.2, 4.3 and 4.4 for further elaboration in the remaining part of this section.

 Figure 4.17 PDP Threshold τ of different Power Dissipation Problem

To study the Power Dissipation Problem generalized from each test set and determine TIP optimizing Scheduling Schemes, the TIP dissipation pattern in each PDP is analyzed. Deriving viable Scheduling Scheme to remedy the TIP based on the given PDP makes the scheduling scheme mapping a linear process and reduces scheduling decision overhead at runtime. All the Power Dissipation Problems (PDPs) categories addressed in this section are derived based on the problems discussed in the literatures cited. Each PDP is further generalized and elaborated based on the analysis of threshold patterns recorded.

Cache misses can be avoided with help from the NUMA structure and predicting memory access patterns [9], however as discussed earlier this section, the miss rate is highly Working Set Size and cache size dependent. As shown in Table 4.2, the Cache Coherence in Multithreading environment is studied to denote a category of cache coherence related Power Dissipation Problems (PDPs):

Table 4.2 Threshold of cache miss dominating Power Dissipation Problems

Power Dissipation Problem	q	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}
Non sequential data fetch	1	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.1	0.3	0.2	0.1	0.1	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.2	0.0	0.2
Inconsistent data layout	2	0.1	1.0	0.0	0.0	0.0	1.2	0.0	0.0	0.0	0.0	0.7	0.3	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.0	0.3	0.0
Larger than cache data sets	3	0.0	0.6	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.2	0.0	1.0	0.1	0.1	0.0	0.1	0.1	0.0	0.0	0.0	0.2	0.0	0.1	0.0	0.0
Cache upgrade request	4	0.0	0.0	1.1	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.6	0.5	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0

PDP 1. Non sequential data fetch: cache prefetcher fetches data in a sequential manner, randomly allocated data causes more cache misses. However, it is difficult to keep data allocation in sequence in SMT CMP architecture, where frequent context switches result in random allocated memory. Besides additional cache miss occurred, excessive icache (Instruction cache) prefetch miss is also generated in this PDP. Thus the corresponding PMP threshold of L1 icache load (τ_7), L1 icache load miss (τ_8), L1 icache prefetch (τ_9) and L1 icache prefetch miss (τ_{10}) etc. are used to identify PDP 1. DEDICATED Scheduling Scheme that is going to be presented in detail later in Section 4.3.2 can be applied to keep the requested memory in nodes with less NUMA distance.

PDP 2. Inconsistent data layout: data layout affects the spatial locality of data stored in memory. Changing the layout of the data (eg. object to column) confuses the prefetcher about its reference memory location. [10] proposes a technique to decrease the influence of spatial locality with reference to single application execution. However in the case of multitasking application execution, data layout varies from app to app and it is more efficient to group apps with similar data layout at runtime to improve the performance of the system. The prefetch miss in dcache (Data cache) is dominating and the L1 dcache miss and L2 cache miss is higher than PDP 1 as PDP 2 is as the Working Set Sizes varies from thread to thread. Thus the corresponding PMP threshold of L1 dcache load (τ_1), L1 dcache load miss (τ_2), L1 icache prefetch (τ_6) and L2 cache load (τ_{11}) etc. are used to identify PDP 2. HOLD Scheduling Scheme can be applied to interleave the application threads with different data layout and decrease the Working Set Size inconsistency in a given sampling period.

PDP 3. Larger than cache data sets: an application that works on Working Set with a size that is larger than the cache sizes generates more cache misses, due to the inconsistency between data requested and cached. Threads belong to the same application are likely to execute the same loop and access data located in the same memory location. In this PDP prefetch miss drops and the TIP is dominated by cache misses. The L2 cache surpasses the L1 dcache miss as the WSS is too large. Thus the corresponding PMP threshold of L1 dcache load miss (τ_2), L2 cache load miss (τ_{12}) etc. are used to identify PDP 3. HOLD scheme is applied to hold the thread working on the large Working Set and executed it at a later sampling interval with a large time slice from the SUPER scheme.

PDP 4. Cache upgrade requests: in MESI protocol, threads requesting to write to cache line inflict extra processor cycles to upgrade itself from shared state. However, the upgrade is unnecessary if the sharing threads are accessing unrelated data stored in cache. Unnecessary upgrades increase the chance of cache miss and write back operation to memory location. In PDP 4 L1 dcache store miss and hits increase as the number of unnecessary upgrades increases. Comparing to PDP 3 the L2 cache miss rate is not as much and yet noticeable. To resolve this problem. Thus the corresponding PMP threshold of L1 dcache store (τ_3), L1 dcache store miss (τ_4), L2 cache store miss (τ_{14}) and L2 cache prefetch miss (τ_{15}) etc. are used to identify PDP 4. DEDICATED scheme is applicable to allocate threads requesting to perform writing action to dedicated CPU cores [11].

In the earlier discussion of this section the impressive amount of TIP dissipated at context switch and thread migration has been addressed, where multiple component in cache and memory hierarchy are flushed and reloaded. As shown in Table 4.3, the context switch and thread migration is studied to denote the category of PDPs that relates to the various system performance overheads in context switching and thread migration [12]:

Table 4.3 Threshold Power Dissipation Problems resulted from context switch

Power Dissipation Problem	q	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}
TLB reload	5	0.0	0.0	0.3	0.1	0.0	0.0	0.0	0.3	0.2	0.0	0.0	0.3	0.0	0.6	0.4	1.1	0.2	0.3	0.6	0.0	0.3	0.8	0.4	0.0	0.0
Cache miss rate increase	6	0.0	0.5	0.0	0.9	0.0	0.0	0.0	0.3	0.0	0.0	0.0	1.0	0.0	0.1	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0
Stalled Pipeline	7	0.0	0.2	0.0	0.2	0.0	1.3	0.0	0.0	0.0	0.3	0.0	0.2	0.0	0.2	0.1	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.2	0.4

PDP 5. TLB entries reload: the time of TLB reload is page size dependent. Hence in the case of concurrently executing applications with varies memory pages of small sizes is likely to result in excessive TLB miss and generate memory page table fault. PDP 5 is dominated by dTLB store and load events, while the L2 cache load

and hits are strongly correlated to the TLB miss and hit counts. Thus the corresponding PMP threshold of dTLB load (τ_{16}), dTLB load miss (τ_{17}), iTLB load (τ_{22}) and iTLB load miss (τ_{23}) etc. are used to identify PDP 5. To overcome this problem DEDICATED and HOLD schemes are applied to set CPU affinity to application threads according to their page sizes as detailed in Section 4.3.2. Hence context switching TLB overheads can be reduced.

PDP 6. Cache miss hit: excessive cache miss rate happens during frequent inter core and even cluster migrations. It results in the highest L2 cache miss rates amongst all PDP and the L1 cache miss count is almost in phase with L2 cache miss. The TLB cache miss is frequent but less than PDP 5. DEDICATED scheme is viable to pin application threads that tend to migrate beyond clusters unnecessarily to the current cluster. Thus the corresponding PMP threshold of L1 dcache load miss (τ_2), L1 dcache store miss (τ_4), L1 icache load miss (τ_8) and L2 cache load miss (τ_{12}) etc. are used to identify PDP 6. At the same time super slice can be applied with increased operating frequency to reduce the migrating tendency. The details of the SUPER scheme is described in Section 4.3.2.

PDP 7. Pipeline stages flush: context switch amongst threads from different applications impact the system performance with additional number of dcache and icache prefetch misses. It is a result of memory structure change at the instance of context switch the branch predictor does not get to rightfully predict the branch according to the historical branching result. Thus the corresponding PMP threshold of L1 dcache prefetch miss (τ_6), L1 icache prefetch miss (τ_{10}) and branch load miss (τ_{25}) etc. are used to identify PDP 7. To remedy this effect of inter application thread switches, the SUPER scheme is applied to extend the quantum of the scheduled application threads. Hence, the number of context switches can be reduced and as well as the miss rate increased at the very moment of the switch. The detailed mechanisms are shown in Section 4.3.2.

Executing applications with Serial bottleneck [13] in its codes in SMT environment leads to thread serialization, where all threads of an application wait for one thread before proceeding further. This defines a category of PDPs as shown in Table 4.4.

Table 4.4 Threshold Power Dissipation Problems due to serial coding bottleneck

Power Dissipation Problem	q	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}
Resource Conflict	8	0.0	0.2	0.1	0.0	0.1	0.0	0.5	0.3	0.1	1.3	0.0	0.1	0.0	0.2	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2
Critical Section Wait	9	0.0	0.2	0.0	0.0	0.9	0.2	0.0	0.3	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.7	0.0	0.4	0.0	0.2	0.2	0.2	0.0	0.1	0.2

PDP 8. Resource conflict: the threads are waiting for I/O resource to be released. This could result in pipeline stall, excessive icache branch miss prediction and branch loading failure. Thus the corresponding PMP threshold of L1 icache load (τ_7), L1 icache load miss (τ_8), L1 icache prefetch miss (τ_{10}), dTLB load (τ_{16}) and branch load miss (τ_{25}) etc. are used to identify PDP 8. DEDICATED scheme is adoptable to allocate all threads from the same application in one CPU core and executed at an increased operating frequency, which is to sacrifice the performance boost from multithreading to achieve less waiting time and context switches amongst application threads.

PDP 9. Critical section wait: a halt in multiple threads caused by a code segment of a thread that cannot be forked to be executed in parallel. Unlike PDP 8, PDP 9 does not generate excessive icache miss and context switch in the CPU core executing the critical code segment. It increases the number of branch load misses in

other CPU cores executing the halted threads. Thus the corresponding PMP threshold of L1 dcache prefetch (τ_5), L1 icache load miss (τ_8), dTLB load (τ_{16}) and dTLB store (τ_{18}) etc. are used to identify PDP 9. The DEDICATED scheme can detect the Critical section wait by predicting the TIP pattern and increase the operating frequency of the core executing the critical code segment to avoid or reduce the time taken in a section wait.

4.3 Structure of the Re Scheduler

In this section the detailed structure and operation of the proposed Re Scheduler are elaborated. The functionality of each function block, the operation sequence of the scheduling schemes and their realization are addressed individually.

4.3.1 Re Scheduler Function Blocks

Application threads are scheduled in a sequence based on their priority and length according to their weight in modern Complete Fair Scheduler (CFS) [116] of Linux system and other heterogeneous architecture targeted schedulers such as the Global Task Scheduler (GTS) [107] and In Kernel Switcher (IKS) [68]. The application threads are queued according to their sequence in a CPU dedicated run queue to wait for their turn to be executed. Every application thread in the run queue gets to be executed according to their predetermined scheduling length, which is named as the time slice length in every CFS scheduling epoch. The Re Scheduler focuses on scheduling application threads according to their Thread Interference Power and Thread Solitary Power (TIP-TSP) relationship determined dynamically at runtime instead of scheduling based on preset priority, thread niceness and power budget of CPU cores as practiced in the conventional schedulers.

Figure 4.18 Re Scheduler scheduling steps and function blocks

Figure 4.18 illustrates the run queue of a CPU core and its content that changes at every scheduling epoch. Each bar () denotes a time slice of the application thread of the same color. Assume that all application threads in the figure has the same time slice length and each epoch is five time slices in length. At the epoch that starts at Time T_1 , there are five different application threads (5 colors) in the run queue waiting to be dispatched into the CPU core. The content in the run queue is then named as $AppSet_1$ that contains application threads from the five different applications. At T_2 the first Epoch is finished and the second Epoch is about to start, the content of the run queue is still $AppSet_1$ as there are still five different application threads even each of the threads has one time slice less of workload. At Epoch T_3 , there are only four application threads in the run queue as one of the thread has finished executing its workload during the Epoch started at T_2 . Thus run queue content is now named differently as $AppSet_2$. As illustrated in Figure 4.18, a five step process has been adopted to analyze the TIP-TSP relationship and make scheduling decision accordingly: 1. several application thread executing variables such as the Performance Monitoring Parameters (PMPs) of the executing $AppSet$ are monitored and updated in the $AppSet$ Table at the end of their executing epoch; 2. the PMP values are converted into Cost in number of cycles according to the PMP quantization equations deduced in Section 4.2.1 as listed the Cost table; 3. the Table Walker function block will Forward Walk the $AppSet$ Table to deduce the member application threads of the $AppSet$; 4. the Table Walker Backward Walks the $AppSet$ Table to determine the Scheduling Scheme parameters those are used to control the sequence and length of the scheduling; 5. apply the Scheduling Scheme with the parameters determined from the Backward Walk to adjust the run queue sequence, time slice length, operating frequency and scheduling CPU core for the targeted $AppSet$.

AppSet Table Structure

To learn the TIP-TSP relationship and the execution pattern of a given $AppSet$, its historical behavior has to be kept in track. The $AppSet$ Table, is the term used to describe the data structure that stores all the relevant information. Figure 4.19 shows the structure of an $AppSet$ Table. Each entry belongs to an $AppSet$. There are nine different parameters under each entry and an $AppSet$ different set of the parameters for a different CPU core.

Figure 4.19 Per core $AppSet$ Table

The *AppSet* entries are updated at the beginning and the end of a scheduling epoch. Two parameters are updated at the beginning of the epoch:

Num_{seen} The first parameter to be updated is the number seen Num_{seen} , which denotes the number of occurrence the *AppSet* in the run queue. The counter is incremented only once before the run queue content or the *AppSet* changes.

T_{lastseen} The second parameter to be updated is the time last seen $T_{lastseen}$ that denotes the value of the time stamp of when the Num_{seen} counter is last updated.

The rest of the parameters are updated at the end of an epoch:

T_{sched} The total time scheduled $T_{sched,T_n} = T_{sched,T_p} + T_{lastepoch}$, denotes the amount of total CPU time in nanoseconds that this *AppSet* has been dispatched from run queue and executed; where T_{sched,T_p} is the total time scheduled prior to the past epoch; and the time last epoch $T_{lastepoch} = T_p - T_n$ denotes the length of the past epoch in nanoseconds, where T_n is the time now at the end of the past Epoch, T_p is the time of the beginning past epoch.

Vol_{total} The total workload volume Vol_{total} that denotes the amount of CPU utility in cycles has been utilized all scheduling epochs by the *AppSet*. It is computed with the equation $Vol_{total,T_n} = Vol_{total,T_p} + Vol_{T_p \rightarrow T_n}$, where Vol_{total,T_p} is the total volume of the workload executed prior to the past epoch; $Vol_{T_p \rightarrow T_n} = f_{T_p \rightarrow T_n} \times T_{lastepoch}$ denotes the workload volume has been executed in the past epoch from T_p to T_n , where $f_{T_p \rightarrow T_n}$ denotes the operating frequency of the CPU core in the past epoch.

PMP_{total} The total Performance Metric Parameters counts PMP_{total} of the given *AppSet* is updated using the following equation:

$$PMP_{total,T_n} = PMP_{total,T_p} \times \left(1 - \frac{Vol_{T_p \rightarrow T_n}}{Vol_{total,T_n}}\right) + PMP_{T_p \rightarrow T_n} \times \frac{Vol_{T_p \rightarrow T_n}}{Vol_{total,T_n}} \quad (4.13)$$

where, PMP_{total,T_n} is the total PMP generated per CPU cycle at the end of the past epoch, PMP_{total,T_p} is the total PMP prior to the past epoch; $PMP_{T_p \rightarrow T_n}$ denotes the PMP counts from `perf` starting from T_p to T_n .

PMP_{TSP} The PMP format Thread Solitary Power PMP_{TSP,T_n} is a combination of PMP_{total,T_n} from its member *AppSets*.

PDP The Power Dissipation Problem *PDP* is derived using PMP_{total} and PMP_{TSP} with the equation (4.11).

RSS The Resident Set Size is retrieved with Linux `ps` command as the worst case Working Set Size for the *AppSet*.

Num_{cs} The number of context switches Num_{cs} denotes the context switch counts in the past epoch, which is the value of context switching event counts in `perf` from T_p to T_n .

As what has been addressed in the TIP-TSP model, an *AppSet* that contains more than one application threads, is able to be decoupled into smaller *AppSets* that spans the same *AppSet* space. Different combination of the smaller *AppSets* generates different Thread Interference Power TIP. Therefore there are mutually unexclusive terms such as Vol_{total} and T_{sched} have to be updated amongst the member *AppSets*, when the monitored *AppSet* gets updated. Besides, the set of member *AppSets* has to be visible in the event of computing the optimized PMP_{TSP} . Hence the *AppSet* hierarchy is predetermined upon constructing the *AppSet Table*. Figures 4.20, 4.21, 4.22 and 4.23 show an example of maintaining *AppSet* hierarchy when establishing *AppSet Table*

The example is based on an empty *AppSet Table* with a reduced number of entries to demonstrate the update through different hierarchy, and a run queue that gets four time equal sized slices executed at any given epoch. Figure 4.20 shows the application thread enters the run queue at Epoch 1 and *AppSet Table* creates an entry for it named $AppSet_1$. The parameter $T_{lastseen}$ and Num_{seen} are updated at the beginning of the epoch and the rest of the parameters are updated at the end of Epoch 1.

Figure 4.20 *AppSet* at the end of the First Epoch

Figure 4.21 shows that there is a new application thread arrived at the run queue and a new entry named $AppSet_2$ is created. In the *AppSet* hierarchy $AppSet_1$ and $AppSet_2$ are denoted as the Class 1 *AppSets* as they refer to one application thread each. As $AppSet_1$ and $AppSet_2$ are sharing the epoch, hence the run queue content is also updated in *AppSet Table* with new Class 2 *AppSet* entry $AppSet_3$. As indicated in Figure 4.21 the other two *AppSets* are the subsets of the member of $AppSet_3$, hence it has both of their indication colors. The parameters of the Class 1 *AppSets* with the same indication colors are related to Class 2 parameters and are simultaneously updated as the Class 2 parameters are. The PMP_{TSP} of $AppSet_1$ and $AppSet_2$ equals to their own PMP_{total} as they are at Class 1. While computing total workload volumes are changed $Vol_{T_p \rightarrow T_n}$, $Vol_{T_p \rightarrow T_n} = f_{T_p \rightarrow T_n} \times \frac{T_{timeslice}}{T_{lastepoch}}$, where $T_{timeslice}$ denotes the length of their time slices in the last epoch. This term decomposes the total workload volume of the Class 2 $AppSet_3$ and updates the member *AppSets* with the decomposed value.

 Figure 4.21 *AppSet* at the end of the Second Epoch

Figure 4.22 shows that there is a new application thread arrived at the run queue at Epoch 3. There are two new entries created upon the arrival of the new application thread, the Class 1 $AppSet_4$ and Class 3 $AppSet_5$. $AppSet_5$ addresses that the epoch is shared by $AppSet_1$, $AppSet_2$ and $AppSet_4$. However there is no new entry created in Class 2 as the only PMP_{total} updated in *AppSet Table* in Class 2 is from $AppSet_3$ and PMP_{total} from lower Class is essential to compute PMP_{TSP} of a higher Class *AppSet*. Hence there is only one *AppSet* member combination to reconstruct that is $AppSet_3$ and $AppSet_4$, such combinations are named as *Path*.

 Figure 4.22 *AppSet* at the end of the Third Epoch

Figure 4.23 shows how is an additional $AppSet_6$ added to $AppSet Table$ at the end of Epoch 4. At Epoch 4 $AppSet_1$ and $AppSet_2$ have finished executing their workload and retired to the wait queue. $AppSet_4$ is the only $AppSet$ left at the end of Epoch 4. However after $AppSet_1$ is retired, the run queue content momentarily constructs a new $AppSet_6$ between $AppSet_2$ and $AppSet_4$, which creates a new entry in the $AppSet Table$. However as the PMP reading sums the events occurred during the epoch, it is not viable to deduce PMP_{total} of $AppSet_6$ as PMP_{TIP} between it and $AppSet_1$ is unknown. Thus there is only one $Path$ for $AppSet_5$ to choose member $AppSets$ at the end of Epoch 4.

Figure 4.23 $AppSet$ at the end of the Fourth Epoch

Cost Table

Prior to walking the $AppSet Table$ to deduce the Scheduling Scheme parameters, the TSP-TIP values in PMP format are converted from number of counts into CPU cost in cycles. Hence the overall system TIP dissipation can be formatted with a summation amongst all PMPs. The TIP and TSP converted are later used to compute overall scheduler energy budget based on energy saved and making scheduling decisions.

As show in Figure 4.24, the Cost Table is computed on a per CPU core basis, as there could be minor differences in CPU cycles generated from PMPs even between cores from the same CPU cluster. The table in Figure 4.24 shows the layout of the Cost Table and the relationship between cost and WSS. The actual values vary from system to system and are omitted from the table. The PMP cost values are derived with the initialization script addressed in Section 4.2.1, and as the Costs number varies across different Working Set Size (WSS) values as previously discussed, the Cost Table therefore contains different set of PMP cost at different WSS values. Upon converting the PMP value into CPU cycles, the Resident Set Size (RSS) which is equivalent to the worst caser WSS of given $AppSet$ is used as the row selector in the Cost Table.

Cost Table	PMP Cost																									
	WSS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2 ⁴																										
2 ⁸																										
2 ¹⁶																										
2 ³²																										

Figure 4.24 Per Core Cost Table

The Cost equation of a given WSS and CPU core is denoted as $\text{Cost}_{wss} = \{c_{wss}^1, c_{wss}^2, \dots, c_{wss}^i\}$, where i is the PMP sequence number that is used to match the cost factor c to the corresponding PMP value with the same sequence number; wss is the WSS. To convert the given PMP Equation (4.14) is used.

$$CC_{TSP} = \sum_{i=1}^{25} \text{Cost}_{wss} \odot \text{PMP}_{TSP} + \text{Num}_{CS} \times \text{Cost}_{wss}^{cs} \quad (4.14)$$

where CC_{TSP} denotes the number of CPU cycles of the Thread Solitary Power (TSP) in the given *AppSet*; Cost_{wss}^{cs} denotes the Cost factor for context switch. The cost of context switch is converted individually due to the fact that partial cost of a context switch is dissipated through different forms of PMPs, including context switch counts in the direct computation of TSP and TIP while updating the *AppSet Table* over counts the impact of a context switch. Hence Num_{CS} is recorded separately for every *AppSet*. $CC_{TIP} = \sum_{i=1}^{25} \text{Cost}_{wss} \odot \text{PMP}_{TIP} + \text{Num}_{CS} \times \text{Cost}_{wss}^{cs}$ is used to convert the Thread Interference Power (TIP) and $CC_{total} = \sum_{i=1}^{25} \text{Cost}_{wss} \odot \text{PMP}_{total} + \text{Num}_{CS} \times \text{Cost}_{wss}^{cs}$ is used to convert the Total Power Dissipated.

At the end of each epoch, the amount of Power dissipation reduced in PMP format is computed using the equation $\text{PMP}_{saved} = \text{PMP}_{total, T_p} \times \text{Vol}_{T_p \rightarrow T_n} - \text{PMP}_{T_p \rightarrow T_n}$. This value is transferred into CPU cycles with the Cost conversion and result in CC_{saved} . The number of cycles saved is used to deduce CPU time n with equation $CC_{saved} = f_{last\ Epoch} \times n$, where $f_{last\ Epoch}$ is the operating frequency of the last epoch. Hence with given operating frequency f , operating voltage V , core temperature and n the Energy saved can then be solved with Equation (4.15).

$$E_{saved} = P_{saved}(V_{last\ Epoch}, Tmp_{last\ Epoch}, f_{last\ Epoch}) \times n \quad (4.15)$$

where P_{saved} is given in Equation (4.9) with per core determined power coefficients. The summation of all energy saved gives the energy budget, E_{budget} .

AppSet Table Walk

The primary functionality of *AppSet Table* Walker is to compute the optimized *path* from the targeted *AppSet* to its all viable member *AppSets* in lower classes. Along the *path* several *AppSet Table* are accessed and utilized to determine essential parameters in scheduling scheme application. The scheduling scheme that is applied to a given *AppSet* is determined by the Power Dissipation Problem as what has been

discussed in Section 4.2.3. The table walk is carried out by the Table Walker function block, which has its steps and utility elaborated in Figure 4.25.

Figure 4.25 AppSet hierarchy of a multi path *AppSet Table*

Figure 4.25 shows the optimized *path* of *AppSet*₅ in a multi *path AppSet Table*. In terms of CC_{TSP} the optimized *path* of $CC_{TSP}(AppSet_5) = CC_{total}(AppSet_3) + CC_{TSP}(AppSet_4)$. Being multi *path* means that there are alternative *Path* or member *AppSets* to reconstruct *AppSet*₅. The alternative *path* in Figure 4.26 is $CC_{TSP}(AppSet_5) = CC_{total}(AppSet_6) + CC_{TSP}(AppSet_1)$. The alternative *path* is not picked as the optimized *path* due to the Walking Constraint is set as the least TSP and it computes a larger TSP for *AppSet*₅. This Walk Constraint is used in the HOLD Scheduling Scheme to denote the *AppSet* is put on hold to be executed after other *AppSets* has finished instead of uniformly sharing the scheduling epoch. In the example given in Figure 4.26 the optimized *AppSet* on hold is *AppSet*₄. Multiple Walk Constraints can be applied for a Table Walk, to satisfy the need of applying multiple Scheduling Schemes on a given *AppSet* during a scheduling epoch. While the SPUER scheme collects T_{sched} of each Class 1 *AppSet* in the *path* to compute the maximum length of its SUPER time slice.

Figure 4.26 Forward *AppSet Table Walk*

Figure 4.27 Backward *AppSet Table Walk*

To dynamically reduce TIP the optimized *path* is computed every epoch, and a walk consists two stages, namely the Forward Walk and the Backward Walk as shown in Figure 4.26 and 4.27. At the Forward Walk stage: the Table Walker traverses through *AppSet* hierarchy from Class 1 members of the given *AppSet*. It then searches in Class 2 for *AppSets* that only contains the found Class 1 *AppSets*. The Table Walker continues the procedure one class higher until it reaches the class of the targeting *AppSets*. Hence at the end of the Forward Walking the Table Walker finds all possible lower class basic member *AppSets* for the given *AppSet*.

At the Backward Walk stage: the Table Walker walks backward, to look for the optimized *path* based on the Walking Constraint. The Walker looks for all possible member *AppSets* one class below, which is Class 2 in the example shown in Figure 4.28. The Walker then sets up a *Path* from the given *AppSet* to its member *AppSets* one Class lower, and denotes them with *Path* $x.y$, where x denotes the sequence number of the *path* and y denotes the class. Afterwards the Walker goes to one class lower to find the lower class members to fill space spanned by the given *AppSet*. The process continues until the Walker reaches Class 1 *AppSets* and then the optimized *Path* is determined based on the CC_{TSP} computed in the two *Path* established.

Multi Core Table Walk

Setting the CPU affinity in application thread is a good measure to avoid inter cluster migration and interference from executing threads from applications with conflicts over computing resources. In order to determine the CPU affinity of a given *AppSet* walking *AppSet Table*, finding an optimized *Path* within only one core is not enough. Hence Table walk across *AppSet table* in different cores for the given *AppSet* is adopted to determine the CPU core that has the best affinity with it.

To compute the CPU affinity for a given *AppSet*, Forward Table Walk is conducted in *AppSet table* of each core. Figures 4.28 and 4.29 show an example *AppSet* hierarchy in Core 1 and Core 2. Assume that *AppSet_a* is given and waiting to be dispatched into the run queue of either Core 1 or Core 2. To make the decision a Forward Walk is conducted. *AppSet_a* and all its relating *AppSets* are found in *AppSet table* of both cores. There are four *AppSets* found relating to *AppSet_a* in each core for the example given. The resulted *AppSets* are then compared,. Only *AppSets* that exist in both cores are accessed with their CC_{TSP} values computed, which are *AppSet_a*, *AppSet_b*, and *AppSet_c* from Core 1 and Core 2. The TSP in the cycles CC_{TSP} is then summed among the accessed *AppSets* within a core. The CPU core that has the lower summed CC_{TSP} is chosen as the core with better CPU affinity to the given *AppSet_a*.

Figure 4.28 AppSet hierarchy in Core 1

Figure 4.29 AppSet hierarchy in Core 2

Next Thread Prediction

While determining Scheduling Scheme decisions such as the length of the SUPER time slice in the SUPER scheme, the *AppSet* on hold in the HOLD scheme and the time of core occupation in the DEDICATED scheme, which thread is arriving at the next epoch is an important factor to consider. For example *AppSet_a* is chosen to be the *AppSet_{hold}* on hold according to the Table Walk result with CC_{TSP} Walk Constrain. However *AppSet_b* is predicted to arrive at the next epoch and the CC_{TIP} between these 2 *AppSets* is larger than the CC_{saved} . Hence the optimized solution will be executing *AppSet_a* in the current epoch and holding another *AppSet*. Figure 4.30 shows that there are three thread related information used in predicting the future thread, the run queue content, thread next predicted and the thread next.

Figure 4.30 Run queue content and prediction of next incoming application thread

The arrival time of a given Class 1 *AppSet* is computed with the equation $T_{next} = \left(T_{lastseen} + \frac{T_{lastseen}}{Num_{seen}}\right) - T_n$, where T_{next} denotes its predicted time until the arrival of the *AppSet*. The next *AppSet* arriving at run queue is $AppSet_{nextarr} = \min AppSet(T_{next})$. Other than predicting the next Class 1 *AppSet* arriving, the presence of *AppSet* in the run queue can be determined by Forward Walking and access $T_{lastseen}$, T_{sched} and Num_{seen} from *AppSets* on the optimized *Path*. With the parameters fetched the time till retiring of the current *AppSets* are computed with $T_{retire} = (T_{lastseen} + T_{sched}) - T_n$. The *AppSets* on the *Path* of the current *AppSet* that remains in the run queue at the next epoch is $AppSet_{nextrem} = \max AppSet(T_{retire})$. Hence the $AppSet_{next}$ in the run queue at the next epoch is likely to be the sum of $AppSet_{nextarr}$ and $AppSet_{nextrem}$.

4.3.2 Re Scheduler Scheduling Schemes

There are three scheduling schemes implemented in Re Scheduler, the HOLD, SUPER and DEDICATED scheme. Which is the same composition of scheduling scheme as proposed in Chapter 3. The aim of the scheduling schemes is to reduce the Thread Interference Power occurred in multithreaded scheduling epochs. Each of the scheduling scheme makes scheduling decisions based on the relevant *AppSet* entries in *AppSet table*. The detailed decision making process for each scheduling scheme is discussed in this section.

HOLD

The HOLD scheme reorders the scheduling sequence of application threads in the run queue according to the Thread Interference Power dissipation level between each combination. In a given high class *AppSet*, One *AppSet* from the lowest class of the member *AppSets* is chosen to be put on hold as $AppSet_{HOLD}$ for a period in time $T_{release}$ that denotes a countdown timer named the time to release. Up on the expiration of the time to release, $AppSet_{HOLD}$ is then scheduled normally. The process used to determine the parameters of $AppSet_{HOLD}$ and $T_{release}$ are the Forward Walking and Backward Walking of *AppSet Table*. The example below shows the detailed steps of parameter determination.

Figure 4.31 shows $AppSet_i$ that consists of four different Class 1 *AppSets* is decomposed in to its Class 1 members at the end of a Forward Walk. Figure 4.32 shows the four *paths* found for $AppSet_i$ through Backward Walking based on *AppSet Table* entries. In each *path*, the lowest Class *AppSet* is chosen to be $AppSet_{HOLD}$, which are Class 1 *AppSets* in the example shown. There are two Walking Constraint applied in the searching of the optimized *path* ($Path_{op}$): CC_{TSP} and T_{retire} of $AppSet_{HOLD}$.

Figure 4.31 Decomposed $AppSet_i$ after Forward Walking

 Figure 4.32 Different path that leads to reconstruct $AppSet_i$ at Backward Walking

The time to retire the $AppSet$ on hold, $T_{retire.HOLD}(AppSet_{HOLD}) = (T_{lastseen} + T_{sched}) - T_n + T_{release}$, where $T_{release} = \frac{T_{sched}(AppSet_{run})}{Num_{seen}(AppSet_{run})}$ denotes that due to being put on hold the $AppSet_{HOLD}$ stays in the run queue for an additional time equals to the average scheduling time of the $AppSets$ that is not on hold and running prior to it. Thus after applying the HOLD scheme the time to retire dependent $AppSet_{next}$ is subject to change from its original structure $AppSet_{next.original}$ to $AppSet_{next.HOLD}$. To determine $Path_{op}$ or $AppSet_{HOLD}$ the following equation is used $CC_{total}(AppSet_{next.HOLD}) - CC_{total}(AppSet_{next.original}) < CC_{saved}(AppSet_{Path_{op}})$, which denotes that the recorded CPU cycles saved from $AppSets$ on the chosen $Path_{op}$ is expected to be more than the additional CPU cycles generated by $AppSet$ structure change from $AppSet_{next.original}$ to $AppSet_{next.HOLD}$.

Implementing Dynamical Voltage Frequency Scaling (DVFS) can avoid the effect of changing in $AppSet$ structure between $AppSet_{next.original}$ and $AppSet_{next.HOLD}$ can be eliminated by removing the additional time term $T_{release}$ in $AppSet_{HOLD}$. The term can be removed by increasing the Operating Performance Point (OPP) with `cpufreq governor` to give the targeting CPU core a boost in operating frequency. The factor of frequency increased is deduced with the equation $\frac{T_{retire.HOLD}(AppSet_{HOLD})}{T_{retire}(AppSet_{HOLD})}$. To justify the OPP adjustment the additional energy dissipation is calculated with the frequency voltage pair at the increased OPP, sampling interval $n = T_{retire}(AppSet_{HOLD})$. The OPP will be increased if the additional energy dissipation is less than the E_{budget} .

SUPER

The SUPER Scheduling Scheme reduced the amount of Thread Interference Power dissipated switching between different application threads at each scheduling epoch. It directly reduces the number of context switches and the numerous number of CPU cycles generated by each context switch. While applying the SUPER scheme, the application threads or the Class 1 member $AppSets$ are given a larger time slice to fill up a scheduling epoch as much as possible to avoid context switching between different threads. $AppSet$ that is given a super time slice is named as $AppSet_{SUPER}$, and the time slice length is denoted with the term $T_{timeslice}$. The example below shows the deduction of super time slice length in the SUPER scheme.

Figure 4.33 shows $AppSet_a$ that is made of three Class 1 member $AppSets$. Assume equal time slice sizes, $AppSet_a$ has a workload size of 8 time slices, $AppSet_b$ has a 6 time slices of workload and $AppSet_c$

has 2. To deduce the amount of time slice extension Forward Walk is conducted, at the end of the Forward Walk, all the Class 1 member *AppSets* are accessed and having their Vol_{total} , T_{sched} , and Num_{seen} . For each Class 1 *AppSet*, their average workload volume per run queue entry in CPU cycles are calculated with the equation $Vol_{per\ sched} = \frac{Vol_{total}}{Num_{seen}}$, and the average workload process per unit time is deduced using $Vol_{ave} = \frac{Vol_{total}}{T_{sched}}$. The equation $Vol_{done} = Vol_{ave} \times T_{elapsed}$ is used to gauge the volume of workload that has been completed so far, where $T_{elapsed}$ is the amount of active scheduled time of the given *AppSet* for its current run queue entry. The remaining workload volume is computed as $Vol_{remain} = Vol_{per\ sched} - Vol_{done}$. The time remained for the given Class 1 *AppSet* is then $T_{remain} = \frac{Vol_{remain}}{f}$, where f the operating frequency of the next epoch. Class 1 *AppSet* with the least T_{remain} is noted as the reference *AppSet* as $AppSet_{ref}$. The super time slice length is either the length of the epoch or the length of T_{remain} of $AppSet_{ref}$.

 Figure 4.33 *AppSet_d* in run queue

Figure 4.34 shows the time slice composition of the given *AppSet_d* with and without applying the SUPER Scheduling Scheme. The figure on the left shows the time slice allocation at every epoch with no super time slice extension. Where the execution between the different application threads switches evenly across the epoch. As a result there are total 7 context switches generated at the end of the third epoch.

 Figure 4.34 *AppSet_d* with normal time slice and with SUPER time slice in each epoch

The figure on the right shows the time slice arrangement with the SUPER scheme applied, where in the first epoch $AppSet_{ref} = AppSet_c$. Hence the super slice length allocated is 2 unit time slices as there is only 2 time slices worth of workload left for $AppSet_c$ to be executed and the epoch has a larger length of 3 unit time slices. In Epoch 1 $AppSet_c$ gets executed first due to the rule that $AppSet_{ref}$ is always the first one to be executed. Hence the run queue does not get hogged by a long time slice that affects the system response time. At the end of Epoch 1 $AppSet_a$ gets executed only half of the super time slice length, which is because that the SUPER scheme automatically expires once the execution of $AppSet_{ref}$ is completed. On Epoch 2 $AppSet_b$ is $AppSet_{ref}$ and $T_{timeslice} = 3$ as $T_{remain}(AppSet_{ref}) = 6$ and the length of epoch is only 3. In Epoch 3 the super time slice length does not change and the SUPER scheme carries on with rotation between Class 1 *AppSets* as the reference $AppSet_{ref}$ is not yet completed. At the end of Epoch 3 only 3 context switches have

been generated under the SUPER Scheduling Scheme while the original time slice allocation generated 7 context switches. The waiting time of $AppSet_c$ in the run queue has been reduced to 2 unit time slices while the waiting time in the original time slice allocation for $AppSet_c$ is 6 unit time slices.

DEDICATED

The DEDICATED Scheduling Scheme reduces the Thread Interference Power dissipation at runtime by assigning the incoming application thread with a CPU core with which it has a better affinity. The CPU core affinity is determined with the multiple page walk process as previously discussed in Section 4.3.1. In a multi-core environment the decision of allocating application threads the run queue of each CPU core is done by the Linux load balancer without much analysis on the interference between different $AppSets$. In the DEDICATED scheme, the TIP generated between the running $AppSets$ in each run queue and the incoming $AppSet_{next}$ is computed and compared to deduce $AppSet_{DEDICATED}$ to execute in the best suitable CPU core. $AppSet_{DEDICATED}$ is the DEDICATED $AppSet$ which denotes the least TIP dissipating *path* for $AppSet_{next}$ in $Core_k^s$, where k is the CPU cluster to which the core belongs and s indicates the sequence number of the core. Figure 4.35 shows an example in a 6 core, 3 clustered CPU environment.

Figure 4.35 DEDICATED Scheduling Scheme execution stages

To apply the DEDICATED scheme to allocate an incoming at the next scheduling epoch $AppSet_{next}$, the run queue content at the current instance $AppSet, T_p$ of all CPU cores is analyzed. In the given example $AppSet_a$ to $AppSet_f$ represent the executing $AppSet$ of each CPU core and $AppSet_g$ denotes the $AppSet_{next}$ waiting for a CPU core allocation. A new $AppSet$ is formed when $AppSet_{next}$ is scheduled to a run queue. Hence possible amount of Thread Interference Power dissipation by the new $AppSets$ is deduced with the Forward and Backward processes. In this 6-CPU core example, six new $AppSets$ are created as $AppSet_h$ to $AppSet_m$ by pegging $AppSet_g$ to $AppSet_a$ to $AppSet_f$ respectively. Each of the new $AppSets$ undergoes Forward Walking in their own $AppSet$ Table. Their respective optimized $paths$ ($Path_{op}$) are then deduced using Backward Walking with the Walking Constraint as CC_{TSP} . However in order to determine which is the most energy efficient core to schedule $AppSet_{next}$, the amount of TIP dissipation with and without $AppSet_{next}$ in each individual run queue is deduced. The original $AppSet$ without $AppSet_{next}$ is denoted as $AppSet_{orig}$ and the new $AppSet$ with $AppSet_{next}$ pegged is denoted as the DEDICATED $AppSet$ or $AppSet_{DEDICATED}$. The amount of speculated CPU cycle difference between these 2 $AppSets$ is computed with the following equation $CC_{total}(AppSet_{DEDICATED}) - CC_{total}(AppSet_{orig}) = f_{next} \times n$, where f_{next} is the frequency value for the respective CPU core to operate and n denotes the sampling interval or named as the length for the CPU to complete that amount of workload with f_{next} . Due to the fact that different CPU core operates at different Operating Performance Points (OPPs) and have different instant temperatures, the number of CPU cycles are then converted into the energy with the given energy equation and parameters of each CPU core. To compute the energy that is going to be dissipated by $AppSet_{DEDICATED}$, the following energy equation is used $E_{core_k^s} = P_{CPU}(f_{core_k^s, T_n}, T_{core_k^s, T_n}, V_{core_k^s, T_n}) \times n$, where $E_{core_k^s}$ is the amount of energy dissipated in a given core s from cluster k ; $f_{core_k^s, T_n}$ and $V_{core_k^s, T_n}$ are the OPP of the given CPU core in the current instance T_n ; $T_{core_k^s, T_n}$ is the current core temperature of the given core; and n is the sampling interval. Hence the value of $E_{core_k^s}$ from each CPU core regarding $AppSet_{DEDICATED}$ is compared, and the CPU core with the least $E_{core_k^s}$ gets to host $AppSet_{DEDICATED}$.

4.4 Multi Cluster Setup

In this section the detailed setup of the multi clustered heterogeneous architecture computing platform is elaborated and the multistage experiment tool chain is addressed.

4.4.1 Three-Clustered CPU Setup

The existing heterogeneous big.LITTLE architecture is well received in commercial products and the research community. It has been proven to be effective in reducing the DVFS overheads [117] by having the freedom of balancing application threads according to their power budget into different CPU clusters. There are normally less threads executed in parallel in the mobile computing environment as comparing to the desktop platform, hence it is viable to exploit the advantage of asymmetrical architecture over the symmetrical

architecture further by having fewer cores in a CPU cluster and more variety of CPU clusters. GEM 5 [96] based simulation is used to justify the possible computation efficiency of the proposed architecture, a mobile computing platform with 3 different CPU Clusters and 2 cores each has been established. ARM Cortex A35, A53 and A57 cores are used in the architecture to form an architecture named as the Small.Medium.Big (S.M.B). The detailed pipeline stages of each modeled CPU is shown in Figures 4.36, 4.37 and 4.38.

Figure 4.36 shows the pipeline stages of a Cortex 35 core which is the most power efficient in-order CPU core out of the three CPUs in the S.M.B setup. It has 8 or 6 pipeline stages according to the type of the instruction, which is the shortest comparing to the other two CPUs. The Small Cluster of the S.M.B consists 2 28nm fabricated A35 cores running in 32 bit mode. The parameters of A35 are retrieved from the released specifications and deduced referencing to the technical manual of the Cortex A7 CPU core.

Figure 4.36 Cortex A35 Pipeline Structure

Figure 4.37 shows the Cortex A53 pipeline stage structures. Similarly to A35 it is also an in-order CPU core. However Cortex A53 has one more decoder than A35, thus it is capable of dealing with dual issue situations and with a tradeoff of dissipating 20% more power than A35 operating at the same OPP. The Medium Cluster in the S.M.B consists of 2 Cortex A53 cores and the modeling parameters of the cores are determined based on the technical manual from ARM.

Figure 4.37 Cortex A53 Pipeline Structure

Figure 4.38 illustrates the pipeline stage structures of Cortex A57. It is an Out-of-Order superscalar CPU core. Therefore comparing to the other two in-order CPU cores it has an additional reorder buffer to facilitate the out-of-order execution of instructions. The A57 has a maximum pipeline length of 23 stages while resolving floating point operations. It is able to decode 3 micro operations (μ ops) and complete 8 μ ops per cycle. As a result, the power dissipation of A57 is the highest among the three CPU cores due to its complicated structures and higher operating frequency. The A57 CPU core model is calibrated according to the ARM technical manual [108]. The detailed CPU core parameters used to model each CPU core of the S.M.B architecture in the GEM 5 simulator is listed in Table 4.5. All three CPU cores are capable of operating in 32 and 64 bit mode. The simulation is based on their 32 bit configuration.

Figure 4.38 Cortex A57 Pipeline Structure

Table 4.5 GEM 5 Simulation Parameters

CPU Blocks			L2 Cache		L1 iCache		L1 dCache		Branch Predictor		TLB		Front-end	Back-end	Pipeline	ROB
Parameters			Size	Associativity	Size	Associativity	Size	Associativity	Entries	Bits	L1 Entries	L2 Entries	Width	Width	Stages	Entries
Cortex A35	1 GHz	2 core 28nm	256 KB	8-way	32 KB	2-way	32 KB	4-way	256	2	10	128	1 μ ops	1 μ ops	6-8	
Cortex A53	1.3 GHz	2 core 28nm	256 KB	16-way	32 KB	2-way	32 KB	4-way	256	2	10	512	2 μ ops	5 μ ops	7-9	
Cortex A57	1.9 GHz	2 core 28nm	2 MB	16-way	48 KB	3-way	32 KB	2-way	512	2	48	1024	3 μ ops	8 μ ops	14-23	128

As the native GEM 5 simulator supports up to only 4 cores in the full system simulation, in order to simulate the proposed 6 core design, the snoop control unit has been modified to bypass the masking of the CPU cores. The simulation consists of two types of CPU cores, the Out-of-Order (O3) cores and the in-order cores, while TimingSimpleCPU model has been used to realize the function of the in-order A35 and A53 cores, O3 model has been used to realize the function of Out-of-Order A57 cores. To facilitate all 6 CPU cores to the system during the simulation, different clock frequencies are assigned to different CPU cores. Multiple L2 caches are added to simulate dedicated L2 cache of each CPU cluster and coherent cross bar is used to maintain the cache coherency amongst the different cache and memory units.

4.4.2 Heterogeneous Oriented Scheduling Approach

To explore the capacity of the big.LITTLE, Linaro has introduced the Global Task Scheduler (GTS) and the In Kernel Switcher (IKS) to govern the thread allocation in between different cores with heterogeneous awareness. As shown in Figure 4.40 [69] the IKS is relatively more straightforward to make scheduling decision across different CPU Clusters.

Figure 4.39 shows that the scheduling scenario in a system with 2 CPU clusters, the big and the LITTLE, and there are 4 cores in each cluster. The IKS constructs 4 cores out of the 8 cores, while each pair consists of 1 big and 1 LITTLE. At runtime only one core out of a pair is active. The IKS decides which CPU core should be active depending on the amount of workload in execution. Once the workload is too much for a LITTLE core to handle, all the application threads will be migrated to the corresponding big core.

The advantage of the approach is simple to implement and little computation overhead while making migration decisions. The disadvantage is that the overhead resulted from inter cluster thread migration can be very expensive. Hence to remedy the migration overhead and unleash the full potential of heterogeneous platform, GTS has been introduced to manage scheduling in big.LITTLE.

Figure 4.39 In Kernel Switcher in big.LITTLE

Figure 4.40 shows the GTS behavior in a big.LITTLE environment. Unlike the IKS, in GTS all CPU cores in every clusters are active at the same time. Given its workload, the GTS decides which CPU core an application thread should be scheduled to, and which core should the thread be migrated to in the case if one or more cores are unbalanced with too much or too less workload.

Figure 4.40 Global Task Scheduling in big.LITTLE

The advantage of GTS is that it can utilize all cores in a heterogeneous CPU at all time. The disadvantage is that the efficiency of the GTS scheduler is number of OPP dependent. GTS saved power dissipation through migrating application thread form a CPU core with one OPP to a core with another OPP. The number of CPU cycles taken for a migration can be less than the sum of time for operating system to make DVFS decisions in a given core. However if there is limited number of different OPPs in a big.LITTLE CPU at runtime the GTS will end up having fewer options to carry out the migration or thread allocation. Hence the disadvantage of the GTS occurs when the run queue of each CPU core is balanced with workload and having less gradient in OPPs.

To overcome the limitation of the existing IKS and GTS, the proposed Re Scheduler is deployed to the Three Clustered S.M.B to justify the effectiveness of being a Thread Interference Power and user application pattern aware scheduler. Re Scheduler is deployed in two versions of Linux: 1. Linux 3.14 to compare with the IKS; 2. Linux 3.10 to compare with the GTS. The detailed setup of the environment is shown in the following section.

4.4.3 Simulation Tool Chain

To conduct an accurate evaluation on the efficiency of the proposed Re Scheduler, full system simulation with power reading and temperature feedback is required as the Thread Interference Power, Thread Solitary Power and the Energy budget computation requires readings from onboard voltage and temperature sensors. However the GEM 5 [96] simulator being a cycle accurate simulator alone is not about to generate detailed power and thermal information. Thus McPAT [97] and HotSpot [57] are employed to provide additional power and thermal feedback with their internal models. The detailed Simulation Tool Chain is as shown in Figure 4.41.

Figure 4.41 Simulation Tool Chain

GEM 5 simulates Linux kernel with Android deployed in the three clustered setup and output the generated traces of execution to McPAT. McPAT sets up its model with the relevant architecture, circuit and tech parameters used in the establishment of the three Clustered environment and computes the power dissipation with the Runtime stats of application execution traces. The power dissipation figures are fed back to the simulated operating system via GEM 5 simulator. The Power figure together with Area and Timing information are feeding forward to Hotspot to generate the Thermal stats for the simulated operating system. Hotspot generates the grid cell based on the Area of each CPU function block provided by McPAT, and use the Grid Cell and the power and timing readings from McPAT to compute the thermal stats using its internal thermal model.

4.5 Results

The proposed Re Scheduler is tested in the GEM5 simulated platform with architecture information shown in Table 4.5. Moby benchmark [102] has been used to simulate usage in a mobile computing environment. The

power parameters are extracted with the McPAT plugin in Thermal design power (W). Three categories of simulation scenarios are created: 1. one app executing individually; 2. two apps executing in pairs; 3. three apps executing simultaneously. There are 8 apps in the benchmark. Hence there are 8 test setups in Category 1, 5 test setups in Category 2 and 4 test setups in Category 3 scenarios. Each test setup consists of ten runs and the percentage of each apps are assigned randomly. The same test set was executed four times with different Linux Kernel schedulers: 1. IKS in Linux Kernel 3.14; 2. Re Scheduler in Linux Kernel 3.14; 3. GTS in Linux Kernel 3.10; 4. Re Scheduler in Linux Kernel 3.10. This is because IKS is only available in Kernel 3.14 and GTS is available in Kernel 3.10. Android Kit Kat was deployed in the experiment.

The Re Scheduler and Re DFN were developed on the ARM architecture with Linux operating system, hence it will be biased to compare its efficiency with Windows schedulers operating on an x86 platform. On ARM architecture, Global Task Scheduler (GTS) and In Kernel Switcher (IKS) are the only two schedulers built for heterogeneous SoCs. Therefore only Linux based GTS and IKS are compared to the proposed Re Scheduler in heterogeneous setup.

The detailed analysis of Category 2 and Category 3 test sets are shown in Figures 4.42, 4.43, 4.44, and 4.45. In each figure, the horizontal axis denotes the miss event terms in the Performance Metric Parameters (PMPs), which are used to gauge the unnecessary power dissipated in the execution of the test set. The other PMPs contributes to the overall CPU utilization which is discussed in the later part of this section. The vertical axis indicates the impact of the amount of CPU cycles generated by given PMP miss event on the total number of CPU cycles elapsed in terms of percentage. The equation $\frac{CC_i}{CC_{CPU\ total}}$ is used to deduce the PMP miss event percentage, where CC_i is the accumulated number of CPU cycles elapsed by PMP miss event i , $CC_{CPU\ total}$ denotes the total CPU cycles elapsed upon completion of the test set. Each figure shows a comparison of the percentage between either the IKS or the GTS and the Re Scheduler.

Figure 4.42 2 App Result comparison between Re Scheduler and IKS in Linux 3.14

Figure 4.42 shows the comparison of Category 2 test set result between IKS and Re Scheduler, the blue colored charts on the top left indicates the result of category 2 test set 1 that executes the Mp3 app and Office

app concurrently. As shown in the chart that the Re Scheduler managed to reduce the amount of additional CPU cycles generated by most of the PMP miss events. However the Re Scheduler generated more TLB prefetch miss as comparing to the IKS, which is due to that the Power Dissipation Problem (PDP) assigned to given *AppSet* during scheduling is PDP 2, the non-sequential data fetch due to the high L1 data cache miss rate. And the Re Scheduler decided to remedy the PDP with the HOLD scheme according to the prearranged mapping between PDP and Scheduling Schemes. However, upon the Office app opening additional documents over the time to stress the system, which resulted in creating numerous small sized memory pages. Hence the applying the HOLD scheme increased the amount of workload from Office app to be executed in one epoch, and generated more TLB prefetch miss as it was more frequent to hit the page boundaries while working with a large amount of small sized memory pages. The DEDICATED Scheduling Scheme can be applied more actively to avoid this situation. This is improved in the work discussed in Chapter 5, where dynamic PDP to scheduling schemes mapping is realized.

The green chart in the top middle indicates the result of Category 2 test set 2 that executes the Twitter and Web app concurrently and the horizontal axis labels apply to all the five charts in the figure. The test results are shown to be dominated by the PDP 6 (the cache miss rate increases) and the PDP 7 (the stalled CPU cycles) as discussed in Section 4.2.3. The Re Scheduler attempts to resolve the issue by applying the SUPER and DEDICATED schemes to interleave application threads executing in the same CPU cores concurrently. However the CPU cycles generated by the PMP miss events are mostly part of the Thread Solitary Power (TSP) instead of Thread Interference Power (TIP), which cannot be remedied using the scheduling schemes, and yet the Re Scheduler tried to seek for better TSP *path* at the *AppSet Table* Walking, and tried different *AppSet_{DEDICATED}*. Thus additional cache misses and migrations occurred comparing to the IKS result. To avoid this problem the overhead of applying the Re Scheduler can be predicted and considered while making scheduling decisions. The problem is addressed and improved as described in Chapter 5.

The orange chart in the top right indicates the result of Category 2 test set 3 that executes the Game and Mp3 app concurrently. The excessive L1 cache misses, TLB miss and Branch prediction miss indicates that *AppSets* in the test set facing PDP 8 (the resource conflict) and PDP 5 (the TLB reload). To resolve the exiting PDPs the DEDICATED and SUPER schemes are used to extend the time slice of *AppSets* and scheduling the application threads that are waiting for the same resource onto the same CPU core. However as the Game app is coded in highly paralleled manner, scheduling all its threads onto the same CPU core resulted in additional instruction cache prefetch miss because the instructions differs from each thread. Hence this problem can be better resolved by applying the DEDICATED to each thread of the app and providing the application with as many CPU cores as possible. The problem is solved in Chapter 5.

The yellow chart in the bottom left indicates the result of Category 2 test set 4 that executes the Shopping and E-mail app concurrently. The increasing L2 cache load miss in this test set denotes that it is dominated by PDP 3 (the larger than cache data set). To resolve this PDP the SUPER scheme and HOLD scheme are applied to avoid scheduling *AppSets* that work on large working sets to be scheduled in the same epoch. However the DEDICATED scheduling scheme is also applied as the energy budget increases, because *AppSet_{HOLD}* and *AppSet_{SUPER}* are scheduled onto idled CPU cores to make up their estimated completion time. Hence the total TIP is reduced at the cost of CPU cycles generated by additional migrations.

The purple chart in the bottom right indicates the result of Category 2 test set 5 that executes the Adobe and Office app concurrently. The result of this test set is also dominated by PDP 3. Unlike the previous test set the TLB load miss increased while applying the Re Scheduler, which is due to the small memory page size of the Office app. However unlike the test set 1, the TLB prefetch miss rate did not increase as there are less over the boundary memory pages, which is due to that the Adobe also generates memory pages in small size. This can be avoided applying DEDICATED scheme to allocate *AppSets* of different applications into different cores. All the problems discussed above are addressed and improved in Chapter 5.

Figure 4.43 2 App Result comparison between Re Scheduler and GTS in Linux 3.10

Figure 4.43 shows the result comparison between the GTS and the Re Scheduler executing Category 2 test sets. The charts are arranged in the same order as Figure 4.42. Generally across all Category 2 test sets, the GTS generates more context switches comparing to the IKS as a result of utilizing all CPU cores at the same time. The PDP that describes the power dissipation in test set 1 becomes data cache dominated PDP 3 instead of being cache prefetch bounded PDP 2, which is due to the fact that given the freedom to move application thread across CPU cores the GTS relocates the application threads according to their working set sizes. As a result the number of migration and data cache store miss due to the changed memory structure increases as comparing to the IKS result. In order to reduce the amount of context switches, the HOLD and SUPER schemes are applied and directly resulted in L1 cache miss reduction. However the work set size increases over time as the *AppSet_{HOLD}* chosen belongs to the Office app that keeps increasing its workload. Thus it resulted in additional L2 cache miss as the working set size exceeds the size of L2 cache more frequently as managed by the Re Scheduler.

The test set 2 result of the GTS demonstrates that PDP 8, where a resource conflict between the Twitter and the Web app, has caused increment in several PMP cache miss events. To resolve this PDP, the Re Scheduler applied the DEDICATED scheme to actively choose the CPU core for *AppSet_{next}*. However the Web app keeps increasing new web pages to stress the system during benchmarking, which makes the prediction

of $AppSet_{next}$ inaccurate, resulting in the Re Scheduler frequently looking for suitable CPU cores as $AppSet_{DEDICATED}$ expires earlier than expected. As a result the Re Scheduler generated more context switches than attempted and this can be remedied by improving the prediction mechanism, and the problem is addressed in Chapter 5.

The test set 3 result of the GTS shows that the PMP cache miss events are a combination of PDP 4 and PDP 8. The Re Scheduler applied the DEDICATED scheme to interleave $AppSets$ with performing frequent cache writing and resource conflicts to avoid unnecessary cache upgrade requests and serial bottlenecks. However it is difficult for the Table Walker to choose $AppSet_{DEDICATED}$ when these two PDP occurs alternatively, while every epoch the resulted $Path_{op}$ is subject to change. Hence the Re Scheduler generated more context switches and data cache miss as compared to the GTS. To remedy this issue, the PDP to Scheduling Scheme matching mechanism is subject to improvement and the mechanism is improved in Chapter 5.

The test set 4 is dominated by the PDP 8 and PDP 5, where instruction cache prefetch miss and TLB miss occurred due to conflict in resources and the additional context switches. Both of the applications are IO bounded and are subject to race for more time slices. Hence the Re Scheduler applied DEDICATED scheme to allocate $AppSets$ into different CPU cores as much as possible. As a result, the instruction prefetch miss is reduced by implementing additional context switches to relocate $AppSet_{DEDICATED}$.

The test set 5 demonstrates a result of combination of PDP 1 and PDP 7, where data cache miss increases due to the cache pollution resulted from context switches. The Re Scheduler utilized the DEDICATED scheme to actively relocate $AppSets$ to reduce the tendency of cache pollution between application threads executed in the same CPU core. However the instruction cache miss event increases in Re Scheduler which is due that the Office app generates additional child threads to execute in parallel, and the DEDICATED scheme tends to group them in one CPU core.

Figure 4.44 3 App Result comparison between Re Scheduler and IKS in Linux 3.14

Figure 4.44 demonstrates the result of 4 test sets from Category 3, which are constructed with 3 apps executed in parallel in Linux Kernel 3.14. Each chart in the figure shows two set of results: the chart in darker color indicates the result of IKS and the chart with lighter color indicates the result of the Re Scheduler. The blue chart on the top left shows the result of test set 1 from Category 3 that contains Mp3, Office and Twitter app. This test set is dominated by the PDP 3, where the large working set size inflicts an excessive amount of L2 cache misses. The Re Scheduler applied the HOLD scheme according to the mapping between PDP and the scheduling schemes, and managed to reduce the working set size of the application threads executing in a run queue consecutively. However the Office app tends to occupy large amount of CPU time for its child threads, hence picking those thread as $AppSet_{HOLD}$ results in additional instruction cache miss. The instruction cache miss can be reduced by giving the Office application threads maximum freedom to be parallelized at the tradeoff of additional scheduling scheme computation overheads. In Chapter 5 the scheduling scheme computation overhead is reduced to reduce the effect of the addressed tradeoff.

The green chart on the top right shows the result of test set 2 from Category 3 that contains Shopping, E-mail and Web app. The result of running it with IKS denotes that it is bounded PDP 1 and PDP 3 as there are excessive L1 instruction cache miss and L2 cache miss. The PDP suggests that the problem is due to inconsistent data layout between different $AppSets$ and their large working set size. The Re Scheduler applied the HOLD and DEDICATED scheme to resolve the problem and successfully suppressed the excessive TIP by inflicting additional context switches.

The yellow chart on the bottom left shows the result of test set 3 from Category 3 that contains Adobe, Game and Mp3 app. The result of executing the test set in IKS shows that it is can be described by PDP 3 and PDP 6, where L1 cache and L2 cache dissipates excessively with a notable amount of context switches. Re Scheduler applied the HOLD scheme to rearrange the run queues, the SUPER scheme to reduce the number of context switches and the DEDICATED scheme to achieve better CPU affinity. However determining $AppSet_{DEDICATED}$ based on the $Path_{op}$ resulted in a tradeoff in the form of the L1 data cache miss event as $AppSet_{HOLD}$ is only determined afterwards. Hence the issue can be solved by considering possible $AppSet_{HOLD}$ in a run queue while determining $AppSet_{DEDICATED}$. The PDP to scheduling scheme mapping is improved in Chapter 5 to realize this solution.

The red chart on the bottom right shows the result of test set 2 from Category 3 that contains Web, Office and Adobe app. The L1 and L2 cache miss event in the result shows that it is dominated by PDP 6. The Re Scheduler applied the DEDEDICATED scheme and the SUPER scheme to avoid unnecessary context switches. The PDP is successfully resolved with minor overheads in L1 dcache prefetch miss.

Figure 4.45 App Result comparison between Re Scheduler and GTS in Linux 3.10

Figure 4.45 shows the result comparison between the GTS and the Re Scheduler executing Category 3 test sets. The charts are arranged in the same order as Figure 4.45. It is observed that in all GTS results, excessive context switches has taken place, which is a result of the dynamic scheduling application threads across all available CPU cores, according to their workload level without considering the interference between different $AppSets$. The higher percentage in L1 and L2 cache miss event also indicates that the GTS is not really Working Set Size aware while making migration decisions. In contrast the Re Scheduler managed to

reduce CPU cycle percentage generated by the context switch and other cache miss event with minimum overheads, which corresponds to the assumption that a scheduler considering the Thread Interference Power and Thread Solitary Power components performs better in a heterogeneous multithreaded environment. The performance gets better as the complexity of the scheduling situation increase.

Table 4.6 summarizes the various PDPs occurred and scheduling schemes applied to each individual test set presented in the test results elaborated above. It is observed from the table that GTS results in more context switch and serial coding bottle neck PDPs (PDP7 stalled pipeline, PDP8 resource conflict and PDP9 critical section wait) comparing to IKS. This is because of the native schedulers (IKS and GTS) are not fully application aware while allocating threads to CPU cores. Threads from different applications may compete for computing resources with each other and threads from the same application may result in internal wait due to the presence of coding bottle necks. All these conflicts occur more frequently in GTS comparing to IKS, which is because the GTS is able to allocate every application thread into any CPU core at runtime. Hence GTS is more likely to create such conflict as a drawback of having higher degree of freedom in CPU core allocation. Re Scheduler utilizes DEDICATED scheme to mitigate the serialized coding bottle neck and able to reduce the TIP generated with better application awareness during application thread scheduling.

Table 4.6 Summary of PDPs occurred and scheduling schemes applied in test sets

Scenario Category	Apps Included in a test set	IKS and RE		GTS and RE	
		PDPs	Schemes	PDPs	Schemes
2	Mp3, Office	PDP2	HOLD	PDP3	HOLD, SUPER
	Twitter, Web	PDP6, PDP7	SUPER, DEDICATED	PDP8	DEDICATED
	Game, Mp3	PDP5, PDP8	DEDICATED	PDP4, PDP8	DEDICATED
	Shopping, E-mail	PDP3	SUPER, HOLD, DEDICATED	PDP5, PDP8	DEDICATED
	Adobe, Office	PDP3	DEDICATED	PDP1, PDP7	DEDICATED
3	Mp3, Office, Twitter	PDP3	HOLD	PDP2	DEDICATED
	Shopping, E-mail, Web	PDP1, PDP3	HOLD, DEDICATED	PDP9	SUPER, DEDICATED
	Adobe, Game, Mp3	PDP3, PDP6	HOLD, SUPER, DEDICATED	PDP3, PDP7	HOLD, DEDICATED
	Web, Office, Adobe	PDP6	DEDICATED, SUPER	PDP8	DEDICATED

Figure 4.46 Simulated result of Re Scheduler in S.M.B CPU core cluster setup

Figure 4.46 shows the average simulation results of the IKS, GTS and Re Scheduler. The results are in CPU cycles collected on a per test set per CPU cluster manner and the average CPU cycles are calculated for each test set category. Hence each chart in a figure represents the total CPU cycle of a test set category (1 app, 2 app and 3 app) executed in a given cluster (S, M and B). The total CPU cycle percentage denotes the amount of CPU cycles dominated by a corresponding PMP event. The CPU utilization indicates the time that a CPU is actively executing workload, which also includes the hit times of the memory units in CPU cycles.

As shown in Figure 4.46, the CPU utilization in GTS and IKS is randomly distributed amongst different Clusters as indicated in the blue colored rectangles. The CPU utilization in Re Scheduler is mostly distributed

in the Small Cluster and least in the Big Cluster as indicated in the red rectangles. It shows that the Re Scheduler prioritizes the cores in the Small Cluster and the Medium Cluster over the cores in the Big Cluster while assigning the workload. In contrast, the GTS and IKS allocate workload according to the current Operating Performance Point in the cores from all the clusters. The CPU core with the best OPP to suffice the power budget of the given workload will be chosen. While the Re Scheduler is able to allocate and contain workloads into the smaller clusters by considering their respective energy budget with thermal awareness. The Re Scheduler achieves better performance over the other two schedulers in three aspects: 1. it generate less CPU cycles in PMP miss events; 2. as the amount of PMP miss event decreased, the CPU cores are subject to more idled cycles that generate less power dissipation; 3. more power are dissipated in the smaller cores as the CPU utilization cycles are actively shifted from the bigger cores. As a result the Re Scheduler achieves less thread migration, more CPU idle cycles and reduced power dissipation.

Figure 4.47 Comparing GTS and Re Scheduler CPU utilization in Linux Kernel 3.10

Figure 4.47 shows comparison between the CPU utilization of the Re Scheduler and the GTS over time in Linux Kernel 3.10. The result is collected with a sampling interval of 1000 ms over a period of 31 seconds. As shown in the figure, the red rectangle encloses the CPU utilization fluctuation level of the Re Scheduler B Cluster executing 2-App test scenario. The blue rectangle shows the level of the GTS B Cluster executing the same test scenario. From the different areas covered by each rectangle, it is observed that comparing to the GTS, the Re Scheduler maintains smoother CPU utilization over time in each CPU cluster by assigning CPU affinity with the consideration and prediction of the workload of a given application thread, which indicates that the Re Scheduler is more power dissipation aware when making scheduling decisions and able to maintain the workload level over a period of time with the energy budget implemented.

Figure 4.48 Comparing IKS and Re Scheduler CPU utilization in Linux Kernel 3.14

Figure 4.48 shows the similar comparison between the IKS and the Re Scheduler in Linux Kernel 3.14. The result indicates that the Re Scheduler has a less fluctuation in between the CPU utilizations in each Clusters, which is due to the fact that there are more TIP-TSP components generated in the GTS compared to the IKS. As the dynamic scheduling of workload across all CPU cores is subject to the generation of more TIP-TSP

components, under the scheduling of Re Scheduler, CPU utilization gap of different clusters are smaller as there are less TIP-TSP components to be exploited during scheduling.

The overall power dissipation achieved by the GTS, IKS and Re Scheduler are shown in Table 4.7 according to different test set categories. The difference in the power reduction across different test set categories implies that the TIP-TSP component increases as the number of apps executed in parallel increases. Hence the Re Scheduler is able to achieve better power reduction in 3 apps category than the GTS. However the power reduction achieved by the Re Scheduler is not as much in Linux Kernel 3.14 as only three CPU cores are considered at scheduling and the TIP-TSP components are fewer in the system with relative simple scheduler decisions to make.

Table 4.7 Power Dissipation Result

Test Set Category	GTS (W)	Re (W)	Power Reduced		IKS (W)	Re (W)	Power Reduced
1 App	1.72	1.489	13.43%		1.57	1.43	8.92%
2 Apps	2.2	1.816	17.45%		1.87	1.78	4.81%
3 Apps	3.92	3.045	22.32%		3.37	3.1	8.01%

4.6 Conclusion

From the result shown in the previous section, it is observed that the Re Scheduler is able to make accurate scheduling decisions based on the Thread Interference Power and Thread Solitary Power (TIP-TSP) model analyzed at runtime, which further indicates that the Performance Metric Parameter based TIP-TSP quantization method is a simple and effective approach for the TIP-TSP determination. The PMP quantization method enables the TIP-TSP to be decoupled into different components and dedicated decisions to be made based on the different patterns observed in them. This has further enabled the mapping between Power Dissipation Problems (PDPs) and the scheduling schemes. Such a mapping has increased the accuracy of scheduling decisions made by the Re Scheduler and reduced the complexity of the decision making problem.

The comparison between the CPU utilization of different schedulers over time has shown that adopting the energy budget based context switching and Operating Performance Point adjustment resulted in a smoother transition of workload in between CPU Clusters and managed to better restrain the workload in the clusters that dissipates more power. The accomplishment is achieved due to the thermal and power budget relationship considered in the energy budget approach, which has provided the Re Scheduler a scheduling constraint that is not only bounded by the performance and power budget of the processor in the current instance but also by the amount of energy the Re Scheduler is able to spend to balance the performance without compromising the overall power dissipated.

However, the TIP-TSP mode and the energy budget based decision making is dependent on the accuracy of the prediction about the next incoming $AppSet_{next}$. It is clearly shown in some of the test cases from Category 2, in a situation where there are less TIP to be remedied, the unpredictable pattern of $AppSet_{next}$ has indeed

reduced the total amount of power reduction and resulted in higher overhead from the scheduling decisions made by the Re Scheduler based on the inaccurate prediction. The impact of less accurate prediction is not as prominent when experienced in Global Task Scheduler and in Category 3 test sets, as the amount of TIP that is deductible in the complex scheduling environment is much more than in the less complex Category 2 test set. Thus the overhead generated from the inaccurate predictions are easily covered. Nevertheless to increase the accuracy of the prediction mechanism and reduce the overhead generated by the scheduler when making scheduling decisions is still one of the main focus of the future studies.

Although the Re Scheduler has demonstrated the ability in better managing a multi clustered asymmetric processor compared to the two native Linux Schedulers, it has also demonstrated that the PDP to the scheduling scheme mapping is too rigid to make dynamic decisions while two or more PDPs exists. Assigning the scheduling scheme to complex PDP situation may result in race conditions amongst the scheduling schemes based on the current static PDP to the scheduling scheme mapping. Different PDPs are resolved by a dedicated scheduling scheme. Different scheduling schemes will game for priority to be applied at the coming scheduling epoch if multiple PDPs are identified and unsuitable scheduling schemes assigned might incur additional TIP dissipation. Hence in the enhancement of the Re Scheduler presented in Chapter 5, the mapping mechanism is extended for better adaptability via the incorporation of neural network based methods.

Chapter 5 Deep Function Network Aided Scheduler

5.1 Overview of Scheduler in DNN Structure

5.1.1 ANN and DNN in Mobile Computing Environment

It has been common for the research community and commercial sectors to tackle complex problems with the help of machine learning and artificial intelligence techniques such as Artificial Neural Network (ANN) and Deep Neural Networks (DNN). Researchers of today have more opportunities than ever to adopt machine learning in their works with a vast diversified houses of applications.

Most of the technology giants not only play the role of being the pioneer of the technology development but have also put in efforts to establish a new horizon of frameworks and tool kits to aid potential developers. For example there are Tensor Flow from Google, Watson from IBM, Wit.ai from Facebook and API package from Microsoft's Bot Framework [151] and Cognitive Services. However most of these machine learning services are cloud based, which restrict the users to have the freedom of accessing artificial intelligence in their personal devices. One of the benefits of having the machine learning applications in the cloud is transferring of expensive computing cost from mobile devices with low energy budget to powerful servers. Thus it could be impractical to apply artificial intelligence on personal handheld computing device as it is subjected to a large amount of power consumption and data transmission in communicating with the machine learning services in the cloud and puts the privacy of the end users at risk.

To mitigate the concerns, hardware manufacturers have worked towards the goal of embedding chips with less power hungry machine learning capability, Qualcomm Zeroth Neural Processing Engine (NPE) [118] is one of the latest achievement out of many attempting projects. The Zeroth NPE makes use of the on-chip Adreno GPU and Hexagon DSP to compute machine learning tasks in a fast and energy efficient manner. The current Zeroth has demonstrated effective usage in obstacle avoidance and malware detection. To embrace the future of personal machine learning, the proposed Re scheduler presented in Chapters 4 is redesigned as a Deep Neural Network, which is able to reduce computing overheads in the general processing unit by exploiting the high Neural Network computing power efficiency in the future Neural Processing Engines.

5.1.2 Reduction of Computation Overhead Generated in Re Scheduler

The utilization of the Neural Network Processing Engine is not the only motivation of redesigning the proposed Re Scheduler in Chapter 4. As discussed Chapter 4, the proposed Re Scheduler chooses scheduling policies based on a dynamically update lookup table *AppSet Table* and matches the corresponded power management scheduling schemes with preset Power Dissipation Problems (PDPs), which has ended up generating unwanted computation overheads and inaccuracy in Performance Metric Parameters (PMPs) prediction as shown in Figure 5.1. The data presented in Figure 5.1 are collected from the simulations performed using the Re Scheduler proposed in Chapter 4. Hence it is assumed that the prediction user usage pattern and

matching the optimized scheduling policy is a nonlinear non-parametric problem that is unlikely to be solved perfectly with limited dynamicity or an existing parametric model. While improving the prediction of incoming events is the main objective of adopting DFN, addressing the computation overhead issue of the Re Scheduler is another objective in Chapter 5, where the implementation of Re DFN is aimed to reduce its computation overhead runtime. This is because Re Scheduler is built on top of the time slice mechanism of currently most computational efficient Completely Fair Scheduler (CFS) that has a scheduling complexity of $O(\log N)$ [116]. Thus reducing its additional computation overhead helps to improve its scheduling efficiency.

Figure 5.1 Scheduling Overhead and Prediction Error of Re Scheduler

The prediction error resulted in performance metrics illustrated in Figure 5.1 has also shown that the traditional parametric lookup table based method fails to predict the result in a fully accurate manner as there are underlying parameters that cannot be monitored. Such parameters are undetectable by the on-chip sensors and are exposed to estimation errors during the process of lookup table updating process. In order to minimize the estimation errors, it is better to adopt a non-parametric method to model the scheduling scheme to PDP mapping problem. Artificial Neural Network (ANN) and Kernel Density Estimation (KDE) can be used to solve problem non-parametrically. ANN has better learning capability comparing to KDE, which enables problems to be solved at runtime with less manual intervention. As mentioned in Section 5.1.1, the dedicated machine learning chips can potentially boost the performance of ANN and further benefit the performance of an ANN based scheduler. Thus, the key to achieve the goal of improving efficiency of the Re Scheduler and solving the complex multicore multithread scheduling problem lies in the reduction of problem complexity. By increasing the modeling details of each individual applications seems impractical in the aspect of providing a generalized solution other than exhausting available system resource at runtime. Therefore, neural network is chosen to reduce the computation overhead of Re Scheduler due to its ability in solving general approximation problems, its capability of computing information unmanned on the fly and its promising future in mobile application. Implementing scheduling decisions using neural network algorithm can help in eliminating the scheduler computation overhead completely [118].

5.1.3 Online and Offline Learning of the Re DFN

In Artificial Neural Network (ANN), there are two different methods of learning, Offline Learning and Online Learning. The Offline Learning is a learning method that the neural network constructs an input independent target function based on a set of pre-collected training data, unlike the Online Learning method whose training data is generated during runtime [119]. The advantage of adopting offline learning in an ANN is that the target function learnt after training can be used as a global approximation, hence requiring less computation overhead at runtime comparing to the Online Learning method. Offline Learning Neural Networks deal less residual error compare to its Online Learning counterpart that has a higher residual error due to more frequent computation of single-step learning. On the other hand the Online Learning method has a faster weight updating rate comparing to the Offline Learning method which guarantees a shorter learning cycle [120].

However in the redesign of the proposed scheduler, it is important to construct the neural network with self-adaptation ability to deal with new unforeseen application patterns and different processor architectures. In order to maintain such adaptability and with as little overhead as possible, both learning approaches has been adopted. The Deep Function Network (DFN) learns in both online and offline manner. The learning is triggered upon arrival of new *AppSet* in the run queue or wait for idled CPU time to train the DFN in small batch offline training.

The Re Scheduler proposed in Chapter 4 is redesigned with neural network components and constructed as a 6-layer Deep Function Network (DFN) named as Re DFN. Each layer of Re DFN is constructed with various neural network function blocks such as Restricted Boltzmann Machine (RBM) and Autoencoder. Each layer of Re DFN maps a part of the function of Re Scheduler as a non-parametric problem and solve it with neural network methods. One drawback of neural network function blocks is the intensive matrix computation cost that aggregates according to the number of neurons and layers of the network. Hence Re DFN is exposed to generate more computation overhead as comparing to the Re Scheduler without the utilization of neural network dedicated chip. Thus neural network optimization is applied to the Re DFN to reduce redundancies in the neural network function blocks to reduce the computation complexity. In the result, the optimized Re DFN has achieved a maximum of 3% computation errors as comparing to the 9% achieved by Re Scheduler. The computation overhead of Re DFN has reduced by an average 46.46% comparing to Re Scheduler. The Re DFN manages to reduce the power dissipation by a minimum of 1.6% and a maximum of 8.1% as comparing to the Re Scheduler.

The main contribution of this chapter includes:

- Redesigned the Re Scheduler with Deep Learning Function Blocks as the Re Scheduler Deep Function Network (Re DFN).
- Optimized the Re DFN performance with several Neural Network optimization methods, including network pruning, weight sharing and matrix compression.
- Assessed the efficiency of Re DFN and Re Scheduler in terms of scheduler overheads and prediction accuracy.
- Managed to achieve scheduler overhead reduction as comparing to the Native Linux In Kernel Switcher (IKS), Global Task Scheduler (GTS) and the proposed Re Scheduler.

5.1.4 Chapter Organization

The rest of the chapter is organized as follows. Section 5.2 discusses the details of the Deep Learning Function Blocks used in the Re DFN. Section 5.3 illustrates the detailed Re DFN structure. Section 5.4 shows the approaches used in optimizing the performance of the Re DFN. Section 5.5 presents the results and the discussions. Section 5.6 concludes the chapter.

5.2 Neural Network Function Blocks and Learning Algorithms

Numerous deep learning methods (e.g., cascaded artificial neural networks that contains more than one hidden layer, each successive layer utilize the output of the previous layer as input.) had been developed over the years, such as the multilayer perceptron (MLP), Stacked Autoencoders (SAE), Deep Belief Networks (DBN), Recurrent Neural Network and the list keeps getting longer every day.

Each of the deep learning methods are developed to resolve a certain type of problem and able to succeed in their designated area of interest. For example the Stacked Denoising Autoencoder performed much better than the Gaussian-Bernoulli Restricted Boltzmann Machine [121]. DBN performed worse than the Deep Autoencoder in speech pattern recognition [122]. Sometimes a collaboration between different deep learning methods managed to boost the performance on certain task. By utilizing Autoencoder for weight initialization in a Restricted Boltzmann Machine (RBM) stacked deep neural network the temporal error in video sampling are greatly reduced [123].

However there is hardly an implementation of deep learning technique in handling real time tasks other than the classical image recognition, human speech recognition and video processing problems. Hence the development of the Deep Neural Network constructed Re Scheduler has taken the consideration of combining different deep learning function blocks built on top of different deep learning methods. Deep Stacking Network [124] is the term used to address a Deep Neural Network facilitating multiple deep learning function blocks such as the RBM and the Autoencoder, especially in large scale problems where the chance of solving the multi aspect problem with a single mean is scarce.

The Re Scheduler makes scheduling decisions and maintains system performance by making multi leveled optimization decisions and predictions, such as feature extraction from time series data, time series arrival prediction, power dissipation problem classification, thermal transient distribution modeling and dynamic runtime temporal redundancy reduction. It is merely impossible for a standalone deep learning method to solve all existing problems with only one neural network.

In the construction of the Re Scheduler Deep Function Network (Re DFN), the basic neural network function blocks chosen are: the Autoencoder that excels while extracting features and predicting time series data; the Restricted Boltzmann Machine [125] which is good at solving stochastic problems and sampling multivariate probabilistic distributions, and also their deep learning variations the SAE and the DBN for solving complex problems.

On the other hand by integrating simpler deep learning blocks, the Re DFN is able to achieve faster training error convergence and less training overheads. The benefits are direct cause of using less layers in a neural network that reduces the vanishing gradient problem [125] and separating inputs to parallel executed function blocks that reduce the matrix size and hence reduce the memory power dissipation.

However there is still a reasonable amount of TIP dissipation cannot be remedied easily at runtime, especially when the computation overhead resulted from the Re Scheduler had a second order temporal impact on the performance status on the system. It will be rather difficult for the neural network to make accurate predictions at runtime. Hence in the Re DFN the term Online Offline Learning Migration has been introduced, where the DFN takes the advantage of batch learning and real time learning. The Re DFN will sample the user app usage behavior to deduce the possible vacant CPU time and conduct batch learning at the vacant CPU time with the training set recorded at the runtime. However the learning batch size is dynamically adjusted as a term of the storage needed for the learning batch, the newly incoming user usage pattern and the TIP dissipation ratio. During offline learning migration, only the system performance monitoring and logging blocks are functioning. Hence it is able for the Re DFN to make decision when it is the right time to trigger and migrate into the online learning mode. By switching between these two learning modes, additional TIP generated from the Re DFN can be very much avoided and highly energy efficient batch learning technique can be applied without compromised much of the optimization provided by the proposed scheduler.

5.2.1 RBM and DBN in Re DFN

The Restricted Boltzmann Machine (RBM) was introduced to model connections between binary pixels and feature detectors by Smolensky in 1986 [126]. Hence the units corresponding to the visible binary pixels and the hidden detectors within a RBM are denoted as visible units and hidden units. As shown in Figure 5.2 a RBM contains two layers, the visible layer and the hidden layer, where each layer has a bias term besides all the fully connected neurons.

Figure 5.2 RBM forward learning in DPD identification sub Layer

A RBM is usually represented as an energy function derived from the Energy-Based Model (EBM) as introduced in [129]:

$$E(v, h) = -b^T v - c^T h - v^T W h \quad (5.1)$$

where v is the vector of visible units, h is the vector of hidden units, b is the vector of bias terms for visible units, c is the vector of bias term for hidden units and w is the weight connecting respective visible and hidden units.

In order to study the weight changing in between two layers, the visible and hidden unit are given the constraint v_j and $h_i \in \{0,1\}$. Therefore the neuron activation functions for visible and hidden units respectively can be derived as suggested by [129] and their probabilistic representation is:

$$p(h_i = 1|v) = \text{sigmoid}(c_i + W_i v) \quad (5.2)$$

$$p(v_j = 1|h) = \text{sigmoid}(c_j + W_j h) \quad (5.3)$$

where sigmoid is the activation function $\frac{1}{1+e^{-n_i}}$, i is the number of hidden units and j is the number of visible units. However the input data used in the proposed scheduler is continuous and it requires additional steps to extend the RBM to handle continuous valued inputs. Instead of sampling visible units in binary state, the expected value of $p(v_j = 1|h)$ is used directly as the current state of the variable [127].

To learn visible units in RBM the maximum likelihood estimator is suggested in [130] and can be applied by substituting the energy function (5.1) into the estimator to derive the probabilistic distribution of v :

$$p(v) = \sum_{h \in \{0,1\}^i} \frac{e^{-E(v,h)}}{Z} \quad (5.4)$$

where, the normalizing factor $Z = \sum_{v \in \{0,1\}^j} \sum_{h \in \{0,1\}^i} e^{-E(v,h)}$ is named as the partition factor in RBM.

To learn the impact of a single sample on the gradient of the connecting weight, gradient ascent is conducted on single log-likelihood sample:

$$\frac{\partial \log p(v^{(\ell)})}{\partial W_{ij}} = \mathbb{E}_{p(h|v^{(\ell)})}[v_i^{(\ell)} h_j] - \mathbb{E}_{p(v,h)}[v_i h_j] \quad (5.5)$$

where, $v^{(\ell)}$ denotes the single sample at ℓ , $\mathbb{E}_{p(h|v^{(\ell)})}$ denotes the data-dependent expectation and $\mathbb{E}_{p(v,h)}$ denotes the model dependent expectation.

Upon determination of the gradient term, the weight W_{ij} can be updated with the following equation:

$$W_{ij}^{(t+1)} = W_{ij}^{(t)} + \eta (\mathbb{E}_{p(h|v) \mathcal{E}(v)}[v_i h_j] - \mathbb{E}_{p(v,h)}[v_i h_j]) \quad (5.6)$$

where, t is the training cycle, η is the learning rate and $\mathcal{E}(v)$ is the empirical distribution of training sample v . In order to update the connecting weight, the data-dependent expectation values and model dependent expectation values have to be computed. In order to reduce the complexity of computation, contrastive divergence has been adopted to compute the expectation of (5.6) instead of using the original Markov Chain Monte Carlo (MCMC) sampling.

In the process of Contrastive Divergence (CD), the RBM visible units and hidden units are updated one after another at the reconstruction sampling phase, as shown in Figure 5.3. The computation complexity of the gradient in RBM comes from the nearly numerous MCMC cycles involved in the sampling for approximation,

whereas in the CD only k steps of MCMC are used to calculate for accurate gradient calculation. This is derived based on the belief that the data-dependent probability distribution $p(h|v)$ moves towards the model probability distribution $p(v, h)$ after k iterations of MCMC, hence $p(v, h)$ should also move towards $p(h|v)$ to speed up the convergence process. Where Hinton [128] has provided proof that it is sufficient and efficient to have the solutions converged with $k=1$. Hence CD with $k=1$ is used in the reconstruction of RBM of the proposed Re DFN layers.

Figure 5.3 RBM Reconstruction DPD identification sub Layer

The RBMs are used as construction blocks in the Re DFN, to facilitate the feature extraction layer, Power Dissipation Problem matching layer, Scheduling Scheme weightage fitting layer and the parameter determination layer due to its probabilistic feature as it has shown better performance in modeling problems with probabilistic model as comparing to other generative and discriminative learning methods [129].

Other than RBM, multilayer Deep Belief Networks (DBN) is also utilized in solving problems with high complexities at the implementation of the Re DFN. DBN is a Deep Neural Network (DNN) structure with stacked RBM to learn deep hierarchical representation of data. As shown in Figure 5.4, there are more than one Hidden Layers in a DBN. Similar to RBM it models a joint distribution between the visible units and the hidden units.

Figure 5.4 Deep Belief Network

The following equation derived in [131] is used to model the probabilistic relationship of DBN between its visible and multilayer hidden units:

$$p(v, h^1, \dots, h^\ell) = (\prod_{j=1}^{\ell-2} p(h^{j+1} | h^j)) p(h^{\ell-1}, h^\ell) \quad (5.6)$$

where ℓ is the number of Hidden Layers, $v = h^0$, $p(h^{\mathcal{K}-1}, h^{\mathcal{K}})$ is the conditional distribution of the visible units conditioned on the hidden units between layer $h^{\mathcal{K}-1}$ and $h^{\mathcal{K}}$, $p(h^{\ell-1}, h^{\ell})$ is the joint distribution of visible and hidden unit in the last RBM of the DBN and $\ell = 3$ as for the case illustrated in Figure 5.4. Unlike a RBM, DBN requires a step named as pre-training to achieve distribution convergence and in Re DFN the greedy layer-wise pre-training method has been adopted. As marked by the dotted grey lines in Figure 5.4 the pre-training works from the visible layer in a bottom up manner. The mean activation of $p(h^1 = 1|h^0)$ is used as the visible unit values of Visible Layer 1 (VL 1) and use the value to train the Hidden Layer 1 (HL 1) as the Visible Layer in a RBM. The result of the training between VL 1 and HL 1 are propagated upwards to train the RBM constructed by HL 1 and HL 2 in the same way. After the pre-training, the DBN will be fine-tuned to optimize hidden layer biases and weights with supervised gradient descent cost function.

5.2.2 Autoencoder and Stacked Autoencoder in Re DFN

The Autoencoder is one of the fundamental unsupervised generative learning models of neural networks, which is widely used to solve Principal Component Analysis (PCA) [79] and sparse coding problems [78]. As the name suggests, the Autoencoder is able to learn higher dimensional features to represent the input data with fewer outputs through the process of encoding.

As shown in Figure 5.5 the main difference between an Autoencoder and a multilayer perceptron (MLP) is that the Autoencoder has the same number of output as its input. An Autoencoder is symmetrically constructed with referencing to the middle layer named Code, the upper or left half of it is the Encoder and the other half is the Decoder. The aim of the Autoencoder is to encode the compressed representation of the input data and decode to reconstruct the input data at the output. In the example given in Figure 5.5, the training data $X = \{x^{(1)}, x^{(2)}, \dots, x^{(t)}\}$, where t denotes the total number of training data sets; $x^{(t)} = \{x_1, x_2, \dots, x_i\}$, where i is the number of inputs variables in one set of training data. The objective function of the given Autoencoder is $x^{(t)} \approx z^{(t)}$, where $z^{(t)} = \{z_1, z_2, \dots, z_i\}$ is the output of network and $t = 10$ in this case. The trigger of neurons in hidden and output layer are denoted with the following equations, $y_j = \alpha^{(2)}(\sum_i w_{ij}^{(2)} x_i + b)$ and $z_i = \alpha^{(3)}(\sum_j w_{ji}^{(3)} y_j + b^T)$, where y_j is the j^{th} neuron of the hidden layer, $\alpha^{(2)}()$ is the activation function of the input layer, $w_{ij}^{(2)}$ is the connecting weight between the input layer and the hidden layer, b is the bias term from the input layer; z_i is the i^{th} neuron in output layer, $w_{ji}^{(3)}$ is the connecting weight between the hidden layer and the output layer which is supposed to be an approximation of $(w_{ij}^{(2)})^T$, b^T is the bias term in hidden layer and an approximated transpose of b .

Figure 5.5 Autoencoder in Classification layer

It is suggested by [81], to ensure the approximation in $(w_{ij}^{(2)})^T$ and b^T , a constraint of $\hat{\rho}_j = \rho$ is implemented. ρ is the sparse factor used to limit the average activation of hidden neuron j ,

$$\hat{\rho}_j = \frac{1}{t} \sum_{i=1}^t \alpha_j^{(2)}(x^i) \quad (5.7)$$

where $\hat{\rho}_j$ denotes the average activation of the j^{th} hidden neuron over t training cycles and $\alpha_j^{(2)}$ is the active function of the j^{th} hidden neuron. To implement the constraint in the cost function, the constraint term is enforced with Kullback-Leibler divergence $\sum_j KL(\rho || \hat{\rho}_j)$ [130]. The overall cost function used in the Backpropagation phase is derived as,

$$J_{sparse}(W, b) = J(W, b) + \beta \sum_j KL(\rho || \hat{\rho}_j) \quad (5.8)$$

where $J(W, b) = \left[\frac{1}{t} \sum_{i=1}^t \frac{1}{2} \|z^{(i)} - x^{(i)}\|^2 \right] + \frac{\gamma}{2} \sum_{l=2}^3 \sum_{i=1}^{10} \sum_{j=1}^3 (W_{ji}^{(l)})^2$ is the original cost function of backpropagation and β is the control factor of the KL divergence sparse term.

Hence the Autoencoder can be trained using the backpropagation algorithm to compute delta error, but at first the average activation $\hat{\rho}_j$ has to be computed with forward propagation of the entire training data set. However the training should be conducted in separated batches if the data set is too large to reduce computation overhead.

To solve problem with higher complexity Autoencoders can be stacked to form a Deep Neural Network named Stacked Autoencoder (SAE). In Figure 5.6 the Full Autoencoder on the left shows an SAE with three hidden layers, a layer-wise pre-training is required to initialize the neural network before starting the training.

Figure 5.6 Stacked Autoencoder

In the layer-wise pre-training, the given SAE will be pre-trained as two separated Autoencoders. The first Autoencoder is formed with the first three layers and features in the hidden layer can be learnt through the encoding function as suggested in [83],

$$z^{(l+1)} = W^{(l,1)}\alpha^{(l)} + b^{(l,1)} \quad (5.9)$$

where $z^{(l+1)}$ denotes the feature in hidden layer of the current Autoencoder separated from the SAE, l denotes the input layer of the current Autoencoder and the sequence number of the layer-wise pre-training, $l = 1$ for the first Autoencoder formed by the first three layers in the example; $W^{(l,1)}$ and $b^{(l,1)}$ denotes the weight and bias term in between the input layer and the hidden layer; $\alpha^{(l)}$ denotes the input layer of the current Autoencoder, while $\alpha^{(1)}$ is the input data.

In the decoding phase, the Autoencoder will be updated with the decoding function,

$$z^{(n+l+1)} = W^{(n-1,2)}\alpha^{(n+l)} + b^{(n-1,2)} \quad (5.10)$$

where n denotes the layer number of the code layer of the current Autoencoder, $W^{(n-1,2)}$ and $b^{(n-1,2)}$ are the reversed connecting weight and bias terms with reference to the encoding function. The same pre-training continues to the adjacent Autoencoders using the output of the previous Autoencoder as input.

In Figure 5.6 the Autoencoder Based Classifier (ABC) on the right shows a case of using SAE for classification operations. The decoder is removed and the output is instead connected to a softmax layer. While conducting pre-training for an ABC the hidden features in the hidden layer of the previous Autoencoder is used as the input of each adjacent Autoencoder. Hence for each pre-training cycle, the layer-wise training advance only two layers in contrast to the three layers advancement in the Full Autoencoder pre-training.

After pre-training, the SAE can be fine-tuned with backpropagation, the computed delta error for each layer in Matrix form is,

$$\delta^{(l)} = ((W^{(l)})^T \delta^{(l+1)}) \cdot f'(z^{(l)}) \quad (5.11)$$

In order to update the weight, the partial derivatives are computed in Matrix form,

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (\alpha^{(n+l)})^T \quad (5.12)$$

where $J(W, b; x, y) = \frac{1}{2} \|y^{(i)} - x^{(i)}\|^2$

The Autoencoder and SAE are utilized in various stages in the Re DFN, such as the PCA sub layer, run queue sequence prediction layer and the feature extraction layer.

5.2.2 Recurrent Neural Network in Re DFN

The Recurrent Neural Network (RNN) is a very well received neural network structure in solving problems regarding sequential data [131]. It is normally used in situation that the current state of the output data is needed to predict the future state of the input data. The RNN also has several variants like Deep Transition RNN and Stacked RNN [93] functioning as generative deep learning methods similar to the SAE and DBN. However the computation of Deep RNN generates undesired overheads that is difficult to ignore at runtime. Hence to keep a compact execution size only conventional RNNs are utilized in the Re DFN.

Figure 5.7 shows a three layered RNN and the influence of its hidden units of the current state over the next state. A RNN updates its connecting weights based on two types of input, the value of the current input at time step t and the value of the previous input at time step $t-1$. At each time step the newly received input value gets integrated in the hidden units and propagated through time to reveal the underlying structure of the data sequence.

Figure 5.7 Recurrent Neural Network Unfolded in time

The computation of hidden units and the output units in Figure 5.7 can be modeled with the equation used in [93],

$$y^t = \alpha(W_{yx}x^t + W_{yy}y^{t-1} + b_y) \quad (5.13)$$

$$z^t = W_{zy}y^t + b_z \quad (5.14)$$

where $y^t = \{y_1^t, y_2^t, \dots, y_j^t\}$ is the hidden layer unit vector at time step t , $x^t = \{x_1^t, x_2^t, \dots, x_i^t\}$ is the input data vector at time step t , α is the activation function, W_{yx} is the Matrix of connecting weights between the input layer and the hidden layer, W_{yy} is the Weight Matrix between the hidden layer at time step t and time step $t-1$, b_y is the bias term of hidden layer, $z^t = \{z_k^t, z_k^t, \dots, z_k^t\}$ is the output layer unit vector at time step t , W_{zy} is the Weight Matrix between the hidden layer and the output layer and b_z is the bias term of the output layer. The $W_{yy}y^{t-1}$ term at the first training cycle is replaced with y^{init} , the initialization bias.

Backpropagation Through Time (BPTT) is used to train the RNN using the method and equations derived in [98]. First the cost of the neural network is defined with the total cross-entropy function $E(z^t, \hat{z}^t) = \sum_t E^t(z^t, \hat{z}^t)$, where $E^t(z^t, \hat{z}^t) = -z^t \log \hat{z}^t$ is the cost function of the RNN at time step t , $\hat{z}^t$ is the output vector predicting the expected vector z^t and the expected vector $z^t = x^t$ in this case.

Just like the original Backpropagation algorithm the BPTT learns the parameters in the RNN with Stochastic Gradient Decent (SGD). The only difference is that the derivative is taken over the sum of the entire training set, $\frac{\partial E}{\partial W_{yy}} = \frac{\partial E^t}{\partial W_{yy}}$, where W is the Weight Matrix. By taking E^3 as an example, its derivative over Weight Matrix W_{yy} will be,

$$\frac{\partial E^3}{\partial W_{yy}} = \sum_n^3 \frac{\partial E^3}{\partial \hat{z}^3} \frac{\partial \hat{z}^3}{\partial y^3} \frac{\partial y^3}{\partial y^n} \frac{\partial y^n}{\partial W_{yy}} \quad (5.14)$$

where $\frac{\partial y^3}{\partial y^n}$ requires to be solved with chain rule and when doing so a summation of gradients from all past time steps are revealed,

$$\sum_n^3 \frac{\partial y^3}{\partial y^n} = \frac{\partial y^3}{\partial y^2} \frac{\partial y^2}{\partial y^1} + \frac{\partial y^3}{\partial y^2} + 1 \quad (5.15)$$

One drawback of the RNN is the Vanishing Gradient problem that occurs after being fed with sequence data with a vast number of time steps, which is equivalent of training a MLP with equivalent number of layers. Hence to resolve this problem the truncated BPTT approach is adopted in the Re DFN, where the training is done every m time steps and the RNN is updated with gradient obtained only within the m time steps. The RNN function block is used to compute the next incoming performance metric sample within the given m sampling intervals to reduce the computation complexity of feature extraction in the first Layer of Re DFN, which is a mini batch training in a sense.

5.3 Layers and Components of the Re DFN

The Re DFN is designed to preserve the problem solving ability of neural networks with low complexity, the purpose of this modification is to improve the overall accuracy of scheduling scheme. The Re DFN inherits the power management scheme mechanisms and scheduling decision making structure of the Re Scheduler. It is not practical to rebuild the Re Scheduler with a single layer neural network due to its complexity. On the other hand, it requires deep neural network with many layers to reconstruct Re Scheduler. Therefore, Re DFN is built with 6 stacked layers of network to realize part of the complex function of Re Scheduler and avoid

making use of a very deep neural network. The Re DFN improves the overall accuracy of scheduling scheme and TIP reduction by eliminating non-parametric estimation errors in $AppSet_{next}$ prediction and solving the nonlinear Power Dissipation Problem (PDP) to scheduling scheme mapping efficiently.

5.3.1 Relations of the Re DFN and the Original Re Scheduler

As shown in Figure 5.8 there are in total six layers in Re DFN and each corresponds partially to a function block in the Re Scheduler. The Re DFN Layer 1 maps the Performance Metric Parameters (PMPs) monitoring function of Re Scheduler by performing features extracting on the time series data created with individual PMPs. While a PMP collected over time is viewed as a time series, it is easier to observe which PMPs that gets dominated by a single Power Dissipation Problem (PDP) reside in the given $AppSet$ over the sampling interval. As the resulted Thread Solitary Power (TSP) of a given Class 1 $AppSet$ is expected to be constant, the TSP component in a PMP time series can be identified as features of different harmonics and used to reconstruct the Class 1 $AppSet$. It is assumed that at the same execution stage of a Class 1 $AppSet$ or an application thread working on similar workload the pipeline usage and PMP generated shows similar patterns as its TSP. Hence the PMP time series are collected and treated as if they are combinations of separable harmonics. The harmonics are extracted through the feature extraction layer and taken as the basic block to identify the observed $AppSet$ and used to study their impact on the system in term of PMPs. There are total 25 input neurons in the first layer and each represents one of the PMPs. The default length of the sampling interval of a time series is ten scheduling epochs, hence there are total 10 readings feeding to the input of Layer 1 each learning iteration, and there is a separated Layer 1 for each CPU core.

Figure 5.8 the Re DFN overview

The features extracted are fed into the second layer of the network, which is the *AppSet* behavior identification layer as shown in Figure 5.8. The second layer functions as the *AppSet Table* updater as in the original Re Scheduler. It is meant to associate the monitored and decomposed performance metrics with *AppSet* in the Linux run queue. The *AppSet* in the run queue will feed its PMP features recorded in *AppSet Table* as bias term to the neural network to indicate the ownership of the given PMP features.

The third layer is constructed to conduct the Power Dissipation Problem (PDP) identification. In the previous iteration of Re Scheduler the identifications process was done by mapping the PDP threshold with the PMP monitored. The Problem identification layer represents 9 different PDPs and each of them is connected to a set of dedicated performance metrics. The Problems identified cover and detect power dissipation surge in PMP format of the current *AppSet* executing in the run queue. The PDP identified belongs to different categories as previously discussed, cache related, coding bottle neck related and migration related. By looking at each PMP monitored the paramount performance metric responsible of each excessive power dissipation surge can be easily identified. And the problem to power dissipation pattern matching can be done without introducing non-parametric estimation errors.

In the fourth layer Scheduling Schemes are picked to mitigate the PDPs identified in the previous layer. This layer functions as the scheduling scheme to PDP mapping in the Re Scheduler. The scheduling scheme picking mechanism in the Re DFN makes use of HOLD, SUPER or DEDICATED scheduling schemes on a per cluster per core basis. The CPU affinity between *AppSet* in run queue and the operating cores, the estimated time to complete the given *AppSet* and *AppSet_{next}* are fed to this layer as bias terms. The bias terms combined with the PDP identification layer output to help to determine which scheduling scheme should be applied in every core.

The fifth layer in Re DFN takes the scheduling scheme picked from the previous layer as input and performs the function of scheduling scheme parameter setting as the Re Scheduler. The bias terms consist of per core temperature, per core CPU Operating Performance Point (OPP), per core *AppSet* run queue, per *AppSet* estimated time to complete and average workload volume of each *AppSet*. The relationship between each bias term and the scheduling schemes are initially statistical, which is the same as what had been implemented in the Re Scheduler. By continuously training the Re DFN the relationship between the bias terms and the output scheduling parameters such as time to retire T_{retire} , super slice length $T_{timeslice}$ and CPU core utilization in energy budget E_{budget} format becomes dynamic. The Learnt Scheduling Parameters then fit the optimization of scheduling schemes at runtime better than the fixed *AppSet Table* based calculation as the non-parametric terms can be eliminated with the help of DFN.

The sixth layer takes the scheduling parameters as the input and its output reassembles the predicted PMP generated by the given *AppSet* after optimization in the next sampling interval. The output neurons in each core are fully connected to each of the scheduling parameter neuron in the same core. The output is then checked against the actual performance metric at sampling interval at the end of the epoch and the delta errors are used to update Layer 3 and Layer 5 of the Re DFN.

5.3.2 Layer-wise structure of Re DFN

Layer 1 Feature Extraction

The input data of the first layer is the Performance Metric Parameters (PMPs) collected with the perf tool Linux extension that includes 25 different variables as shown in Table 5.1. The data is collected at a sampling rate of 10 times per second and each of the training epoch covers the PMPs collected within one second that is 10 sets of 25 variables. Hence the accumulation of variables is regarded as 25 time series data with 10 intervals. The purpose of Layer 1 is to extract features from each variable time series and it can be modeled as solving a feature extraction problem of sequential data. In this problem let input be $X^t = \{x_1^t, x_2^t, \dots, x_n^t\}$, where $n = 25$ is the sequence number of the PMP variables and $t = 10$ is the number of sampling per second. (This t setting is used in the online learning mode. During the offline learning mode of the Dynamic Learning Migration, the sampling frequency is once per second. Hence the training set size for the offline learning is 3.5 MB per hour, and 2 MB after Sparse Matrix Storage.)

Table 5.1 Performance Metric Parameters

L1 dCache Metric		L1 iCache Metric		L2 Cache Metric		dTLB Metric	
1	L1-dcache-loads	7	L1-icache-loads	11	LLC-loads	16	dTLB-loads
2	L1-dcache-load-misses	8	L1-icache-load-misses	12	LLC-load-misses	17	dTLB-load-misses
3	L1-dcache-stores	9	L1-icache-prefetches	13	LLC-stores	18	dTLB-stores
4	L1-dcache-store-misses	10	L1-icache-prefetch-misses	14	LLC-store-misses	19	dTLB-store-misses
5	L1-dcache-prefetches			15	LLC-prefetch-misses	20	dTLB-prefetches
6	L1-dcache-prefetch-misses					21	dTLB-prefetch-misses
iTLB Metric		Branch Metric					
22	iTLB-loads	24	branch-loads				
23	iTLB-load-misses	25	branch-load-misses				

The PMP time series collected is a reflection of the parallel processing application threads in the computing platform. In other words the time series of each PMP variable $X_n = \{x_n^1, x_n^2, \dots, x_n^t\}$ is the resultant outcome of different applications with class $A = \{1, 2, \dots, \alpha\}$, where α is the number of applications that had shared time slice through the CPU pipeline in the past sampling interval of length t . (By assuming the PMP figures calculated reflect the difference between the PMP of the current interval and the previous interval allows the sampling interval to take any length, however longer sampling interval translates to more complicated PMP to *AppSet* relation, hence a smaller sampling is still a better choice.) Each of the PMP variable time series is related to the class by a Feature Matrix of $F_{\alpha,n} = \{f_{1,n}, f_{2,n}, \dots, f_{\alpha,n}\}$, where f_{α} is the Feature Vector with t entries and $\sum_{\alpha} F = X_n$.

As shown in Figure 5.9 a three layered DBN has been employed to extract the features from the sequential PMP time series. Each PMP variable time series undergoes a separated neural network for feature extraction operation, hence there are 25 DBNs in Layer 1. The Hyperparameter of the DBN is listed in Table 5.2 which is determined with layer-wise pre-training and validated with regularization.

Figure 5.9 Function blocks of layer 1

Table 5.2 Hyperparameters of Layer 1

	Number of Neurons	Activation Function	Weight Matrix Size	Input Size
Hidden Layer 1	1	Sigmoid	2×1	10×1
Hidden Layer 2	2	Sigmoid		Output Size
Visible Layer	3×3	Sigmoid	$3 \times (2 \times 3)$	$3 \times (10 \times 3)$

In the DBN, the input $h_i = x_n^t$, one every learning epoch one input value is goes to hidden layer 1 in forward propagation. The number of hidden units in the second hidden layer is set to match the number of nodes in the first hidden layer. To have a second hidden layer that is larger than the input and the output brings sparser Euclidian distance in between the output feature [132]. There are three separated output layers at the visible layer each sharing the same Weight Matrix from hidden layer 1 to hidden layer 2 and the output from hidden layer 2. The purpose of the three separated output layers is to limit the number of features to match α , the number of application thread running in parallel. Each of the output layers is given a bias term $B_\alpha = \{b_1, b_2, \dots, b_\alpha\}$ that is derived from the *AppSet* that contains the application thread. (The thread running in parallel is limited to three which is due to the fact that at such a short sampling interval not many applications will be sharing the time slice and user initiated applications are the only target to the Re DFN). The output layers in visible layers will be only triggered if the bias term is on, hence there will be less features extracted in the scenario of pipelining application threads are less than three. There are three visible units in each output layer to ensure a degree of freedom in the feature selection process later.

Layer 2 Feature Selection

Layer 2 contains two sub layer, the Feature Selection sub layer and the Trend Prediction sub layer. The function of Feature Selection sub layer is to associate the features extracted from input data in Layer 1 with the applications executed and update the relevant entries in the *AppSet Table* according to the application class term α . The feature selection problem can be modeled as a Sparse Coding problem to identify the constrained feature space. The feature vector of application class or *AppSet* α at PMP n is $f_{\alpha,n}$, while the output of Layer 1 is a matrix $V_{\alpha n} = \{v_{1kn}, v_{2kn}, \dots, v_{\alpha kn}\}$ consists α basis vector $v_{\alpha kn}$ to reconstruct $f_{\alpha,n}$. In order to reduce the computation complexity and feature space sparsity, Sparse Coding is conducted to constrain the features

selected into one single vector and use the selected vector to update TIP and STP of the relevant *AppSet* in the *AppSet Table*.

The Sparse Coding operation is realized with a 3-layer SAE with a softmax output layer as shown in Figure 5.10. Each vector of matrix V_{an} generated from Layer 1 is used as value for input layers, and goes through the SAE in forward propagation for ten epochs. The feature basis vectors go through the first Autoencoder for encoding and decoding process. The purpose is to increase the distance between the maxima and minima of the basis vector by removing trivial values. The output value of hidden layer Z_k is fully connected to a softmax layer with two units. After layer-wise pre-training of SAE and back propagated fine tune, the classification neural network obtains the maximum probability of $P(f_{\alpha,n} = 1 | AppSet_{\alpha})$ when the extracted feature belongs to $AppSet_{\alpha}$. The value of softmax layer with larger probability is then updated in the *AppSet* table. Table 5.3 lists the Hyperparameters of Layer 2.

Figure 5.10 Function blocks of layer 2

Table 5.3 Hyperparameters of Layer 2

	Number of Neurons	Activation Function	Weight Matrix Size	Input Size
Input Layer J	3	Hyperbolic	2×3	10×3
Hidden Layer Y	2	Hyperbolic		Output Size
Hidden Layer Z	3	Hyperbolic	3×2	
Output Layer S	2	Softmax	2×3	10×2
Input Layer R	1	Hyperbolic	10×10	Input Size
Hidden Layer H	1	Hyperbolic		10×1
Output Layer O	1	Hyperbolic		Output Size
				10×1

The function of the Trend Prediction sub layer is to predict the next incoming application thread or the *AppSet* in the run queue by analyzing the input data sequence and compare the sequence with the *AppSet Table* entries. The trend prediction can be modeled as time series prediction problem, with a known time series $T_{(1,t)}$ to predict the future trend of sequence $T_{(t+1,t+w)}$, where $(t+1, t+w)$ denotes the sampling window of the time series, from $t+1$ to $t+w$. Variable w denotes the prediction window, which is used to mark the length of predictable cycles after t cycles of learning. The value of the known time series used in this trend prediction problem is the feature vector $f_{\alpha,n}$ of length t . The prediction window of $w = 1$ is used to reduce the computing complexity. Regardless of the small predicted window, *AppSet* sequences are accurately predicted with the vector space mapped with w contains 25 entries shown in Figure 5.11. Figure 5.11 shows the layout of

PMP parameters in an *AppSet Table*. The actual values change at system runtime according to the inputs and the responses from the neural network. The entries in the table are omitted.

Figure 5.11 PMP entries in *AppSet Table*

The trend predicting function is realized with a 3-layer RNN [133] as shown in Figure 5.10. The RNN takes the same input as the first output unit in the SAE softmax layer. As the class 1 in the softmax layer is served as the principal value and the class 2 is used as the tolerance value. The root-mean-square value of the two values is taken and the absolute difference between class 1 and class 2 is used to set the margins for the corresponding value recorded in *AppSet Table*. The RNN is able to predict the feature of next sampling interval with the given data sequence. The output of all RNNs in Layer 2 are used to conduct a row search in the PMP entries of the *AppSet Table*, to identify the next *AppSet* incoming. Although there is only one neuron in the hidden layer of RNN, it will construct a 10×10 weight matrix after 10 iterations. The bias term and the PMP normalization: Each Bias term b_α of $B_\alpha = \{b_1, b_2, \dots, b_\alpha\}$, is a vector consisting t entries of value denoting the possible high and low of feature n of application α , adding bias term to the value of h_j in layer 1 ensures the input features are extracted according to the composition of the *AppSet Table* entries. The bias b_α is the normalized TSP of the *AppSet*. The input PMP data are normalized with McPAT power model to reflect similar power consumption impact based on the PMP readings.

Layer 3 Power Dissipation Problem Mapping

This layer compresses the representation of the Performance Metric Parameter (PMP) into a principal indicator which is used to identify the Power Dissipation Problem (PDP). (The PMP to PDP association is done in a fully connected manner, but the bias terms are assigned during the initialization according to the PDP table for better result.) A PDP has to be generalized from a set multivariate input data which is the PMP in the context of this study. The first step of the generalization is the identification of the principal component from the PMP data set. This problem can be modeled as a Nonlinear Principal Component Analysis (NLPCA) [134] with the assumption that there is a multivariate data space $\mathbb{D}$ contains known variables and an indicator space $\mathbb{F}$ that is a sub space of $\mathbb{D}$. Through the operation of NLPCA the unknown indicator space $\mathbb{F}$ and its identity mapping to the data space is constructed. The first half of the mapping consists of a compression function ϕ_{comp} that reduces the dimensionality of data vector $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_i\}$, where i is the number of dimensions of data space $\mathbb{D}$ and transforms the data vector into the indicator vector $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_k\}$, where k is number of dimension of the indicator space and $k < i$. The other half of the mapping consists of a reconstruction function ϕ_{recon} that reverses the compression process and improves the compression mapping by backpropagating the error term.

The PCA model correlates with the structure of a bottlenecked SAE, where the encoding and decoding layers are able to realize the function of the compression and reconstruction function. The indicator space is also identical to the code layer at the center of the SAE. In order to maintain the nonlinearity of the problem, the SAE is structured with three hidden layers as shown in Figure 5.12. The input is the vector $x_n = \{x_n^1, x_n^2, \dots, x_n^t\}$ consisting the value of a PMP with sequence number n over the sampling interval from 1 to t . The SAE is pre-trained in a layer-wise manner with the input vector (a training set consists of multiple x_n is computed in the Offline Learning epoch and only one vector is used for mini batched Online Learning.) and fine-tuned with backpropagation.

Figure 5.12 Detailed layout of Layer 3

Table 5.4 Hyperparameters of Layer 3

	Number of Neurons	Activation Function	Weight Matrix Size	Input Size
Input Layer $\mathcal{D}$	10	Hyperbolic	10×5	1×10
Hidden Layer $\mathcal{E}$	5	Hyperbolic	5×1	Output Size
Hidden Layer $\mathcal{F}$	1	Hyperbolic	1×5	
Hidden Layer $\mathcal{G}$	5	Hyperbolic	5×10	1×10
Output Layer $\mathcal{H}$	10	Hyperbolic		
	Number of Neurons	Activation Function	Weight Matrix Size	Input Size
Input Layer $\mathcal{I}$	25	Hyperbolic	9×25	1×25
Hidden Layer $\mathcal{J}$	9	Hyperbolic	25×9	Output Size
Output Layer $\mathcal{K}$	25	Hyperbolic		

However as a dimensionality reduction problem under NLPCA model, the decoding layers are only existed for the purpose of Backpropagation learning. The output of the code layer $\mathcal{F}_k$ is used as the input for the following linear PCA layer constructed with a 3 layer Autoencoder. The 3-layer Autoencoder is used to solve

the problem of representing the PMP values of the system with only a few Power Dissipation Problems (PDPs). It is assumed that the relationship within the PMP values and the PDP is linear. Hence a 3-layer Autoencoder is used to solve the linear PCA. Similar to the 5-layer SAE, only the code output is used to denote the PDP identified. To denote the PDP problem with distinction, each neuron in hidden layer J_j is complimented with a bias vector b_j to filter out trivial PMP inputs for problem identification. There are total 9 PDPs as shown in Figure 5.13. The PDP table is updated dynamically with the backpropagated bias error during the training phase of the Autoencoder. Figure 5.13 demonstrates the layout of PDP biases in the PDP table and their relationship with each PMP. The actual reading varies from case to case and are omitted from the table.

Power Dissipation Problem	b_j	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}
Non sequential data fetch	b_1																									
Inconsistent data layout	b_2																									
Cache upgrade request	b_3																									
Larger than cache data sets	b_4																									
TLB reload	b_5																									
Cache miss rate increase	b_6																									
Stalled Pipeline	b_7																									
Resource Conflict	b_8																									
Critical Section Wait	b_9																									

Figure 5.13 PDP Table bias term entries

Layer 4 Scheme Mapping

The function of this layer is to map the Power Dissipation Problems (PDPs) generated from Layer 3 with the scheduling schemes provided by the proposed Scheduler to remedy possible Thread Interference Power (TIP) created in a normal scheduling sequence. As discussed in the previous chapter, mapping the PDP with one scheduling scheme discriminatively does not promise an acceptable TIP reduction at runtime. In order to avoid the race conditions between application threads with statically assigned scheduling schemes, it is suggested that applying a combination of different scheduling schemes on one *AppSet* results in better TIP reduction. Hence the scope of designing Layer 4 is to enable the Re DFN to derive a reasonable combination of scheduling schemes for a given *AppSet* at runtime in an unsupervised fashion with minimum user intervention.

Similar to what has been done for the Re Scheduler, scheme mapping is determined based on the PDP identified and updated in the *AppSet Table*. In order to generate a scheduling interval with more than one scheme applied, the mapping result is expected to be presented in probabilistic format instead of being a binary classification. Each scheduling scheme is assigned with a scheduling interval proportional to their probability of being mapped against the current *AppSet*. Upon confliction the scheduling scheme with high probability has the priority in taking over the scheduling interval. Hence this problem is modeled with a generative approach to discuss the joint distribution between two sets of data, the PDP J_j of the *AppSet* waiting in the run queue Q and the set of scheduling schemes $S_w = \{S_1, S_2, \dots, S_w\}$, where $w = 3, S_1 = HOLD, S_2 = SUPER, S_3 = DEDICATE$.

As shown in Figure 5.14, a RBM is used to construct Layer 4 as it is generative and derives joint distribution natively [74]. The PDP J_j serves as the input of the RBM from Layer 3. The RBM is updated with Contrastive

Divergence k (CD_k), where $k = 1$. In terms of neural network the result generated from the RBM can be regarded as a concise feed-forward neural network that maps the input visible node observation with a possibility of an expected hidden node. A bias vector term is added to the hidden nodes to justify scheduling scheme selecting criteria other than the PDP. Table 5.5 lists the Hyperparameters of Layer 4.

Figure 5.14 detailed layout of layer 4

Table 5.5 Hyperparameters of Layer 4

	Number of Neurons	Activation Function	Weight Matrix Size
Visible Layer v_p	9	Sigmoid	3×9
Hidden Layer h_s	3	Sigmoid	

As shown in Figure 5.15, the conditional bias terms in the visible layer of RBM will be triggered up on the blocking conditions. Once the condition is met, the corresponding scheduling scheme will be constrained on neuron activation and rejected from being chosen as one of the scheduling schemes to optimize the Thread Interference Power (TIP) of the current *AppSet*. The next *AppSet* condition is derived from the trend prediction function block in Layer 2, the SUPER scheme is blocked upon the condition because having similar *AppSet* waiting in the run queue will result in an *AppSet* hogging the time slice through SUPER Scheduling Scheme. The QoS denotes the estimated completion time of the *AppSet*. The HOLD scheme is not allowed to be applied to the *AppSet* if the estimated completion time is violated. The Available Core condition denotes the availability of vacant CPU cores within the same or other clusters of the multi-core platform, the DEDICATE Scheme is disabled in the case of having zero vacant core since there is no freedom in redistributing congested *AppSet* into a sparser core.

Scheduling Scheme	b_{α_s}	Next <i>AppSet</i> (O_k)	QoS (α)	Available Core
HOLD	b_{α_1}		Block if $QoS(\alpha)$ expires	
SUPER	b_{α_2}	Block if $O_k^t = O_k^{t+1}$		
DEDICATE	b_{α_3}			Block if $Core_{free}^t = 0$

Figure 5.15 Scheduling scheme mapping bias term entries

Layer 5 Scheduling Scheme Parameter Setting

The function of Layer 5 is to set scheduling scheme parameters by associating scheme picked in Layer 4 with their respective parameterization constraints in an unsupervised manner. Several optimization decisions have to be made prior to applying the scheduling schemes picked in Layer 4 to targeting *AppSets*, which includes trigger conditions such as *maximum* scheduling period, the minimum scheduling period and the

targeted *AppSet* of the respective scheme. To realize the function a sparse Autoencoder is constructed in an associate setup as shown in Figure 5.16. The Autoencoder consists of three layers and the output is a softmax function. Table 5.6 lists the hyperparameters of Layer 5.

Figure 5.16 Detailed layout of Layer 5

Table 5.6 Hyperparameters of Layer 5

	Number of Neurons	Activation Function	Weight Matrix Size	Input Size
Input Layer $\mathcal{L}$	1	Hyperbolic	$3 \times (9 \times 3)$	3
Hidden Layer $\mathcal{M}$	3	Hyperbolic	$3 \times (2 \times 3)$	Output Size
Output Layer $\mathcal{N}$	1	Softmax		9×1

The output $h_{as}(J_j)$ from Layer 4 is used as the input of each Autoencoder, where α denotes the *AppSet*, s denotes the scheduling scheme and J_j is the PDP derived from Layer 3. There are totally 3 parameters to be set for each of the 3 scheduling schemes, and each of the Autoencoders is responsible for computing one of them. Each neuron in the hidden layer of the Sparse Autoencoder is associated with a conditional bias term $b_{as\beta j}$, where β denotes the scheme parameter and j denotes the sequence of the constraint. The bias is used to constrain the output to the respective conditions of their representing parameter. Figure 5.17 shows the bias terms used to constrain the output and their functions.

Scheme Parameter	$b_{as\beta j}$	HOLD Scheme ($s = 1$)	Scheme Parameter	$b_{as\beta j}$	SUPER Scheme ($s = 2$)
Max Scheme Duration	$b_{\alpha 111}$	$AppSet_{next} = AppSet_{HOLD}$	Max Scheme Duration	$b_{\alpha 211}$	$AppSet_{superSlice}(HV) > T_{remain}$
	$b_{\alpha 112}$	$AppSet_{HOLD}(HV) > T_{remain}$		$b_{\alpha 212}$	$AppSet_{next}(HV) > AppSet_{SUPER}(HV)$
	$b_{\alpha 113}$	$AppSet_{HOLD}(T_{comp})$		$b_{\alpha 213}$	$AppSet_{smallSlice}(T_{comp})$
Min Scheme Duration	$b_{\alpha 121}$	$Overhead_{HOLD}(AppSet_{HOLD}) < CC_{saved}$	Min Scheme Duration	$b_{\alpha 221}$	$Overhead_{preemptive}(AppSet_{SUPER}) < CC_{saved}$
	$b_{\alpha 122}$	$Overhead_{ADD}(AppSet_{HOLD}) < CC_{saved}$		$b_{\alpha 222}$	$Overhead_{reslice}(AppSet_{SUPER}) < CC_{saved}$
	$b_{\alpha 123}$	$Overhead_{FE}(AppSet_{HOLD}) < CC_{saved}$		$b_{\alpha 223}$	$Overhead_{FE}(AppSet_{SUPER}) < CC_{saved}$
Scheme Duration	$b_{\alpha 131}$	$AppSet_{HOLD}(TIP_{generate}) < AppSet_{ave}(TIP)$	Scheme Duration	$b_{\alpha 231}$	$AppSet_{SUPER}(TIP_{generate}) < AppSet_{ave}(TIP)$
	$b_{\alpha 132}$	$AppSet_{HOLD}(TIP_{generate} < TIP_{reduce})$		$b_{\alpha 232}$	$AppSet_{superSlice}(HV) > AppSet_{smallSlice}(HV)$
	$b_{\alpha 133}$	$AppSet_{HOLD}(HV) < AppSet_{ave}(HV)$		$b_{\alpha 233}$	$AppSet_{smallSlice} = AppSet_{next}$
Scheme Parameter	$b_{as\beta j}$	DEDICATED Scheme ($s = 3$)			
Max Scheme Duration	$b_{\alpha 311}$	$Core_{DEDICATED}(E_{budget}) < Core_{DEDICATED}(E_{consumed})$			
	$b_{\alpha 312}$	$Core_{other}(TIP_{generate}) > Core_{DEDICATED}(TIP_{reduced})$			
	$b_{\alpha 313}$	$AppSet_{DEDICATED}(HV) > AppSet_{ave}(HV)$			
Min Scheme Duration	$b_{\alpha 321}$	$Overhead_{migration}(AppSet_{DEDICATED}) < CC_{saved}$			
	$b_{\alpha 322}$	$Overhead_{reservation}(AppSet_{DEDICATED}) < CC_{saved}$			
	$b_{\alpha 323}$	$Overhead_{FE}(AppSet_{DEDICATED}) < CC_{saved}$			
Scheme Duration	$b_{\alpha 331}$	$AppSet_{DEDICATED}(TIP_{generate}) \ll AppSet_{ave}(TIP)$			
	$b_{\alpha 332}$	$AppSet_{DEDICATED}(Core_{other}(TIP) > Core_{DEDICATED}(TIP))$			
	$b_{\alpha 333}$	$Overhead_{migration}(AppSet_{DEDICATED}) > AppSet_{DEDICATED}(TIP_{reduce})$			

Figure 5.17 Scheduling Scheme Parameter mapping bias term entries

HOLD Scheme:

The max duration constraint for the HOLD scheme defines the maximum scheduling duration or the release condition of the $AppSet_{HOLD}$ ($AppSet$ on hold, the $AppSet$ that is put on HOLD while applying the HOLD scheme) to avoid additional TIP generated by holding an $AppSet$ for too long. There are three constraints under this category: b_{a111} , the $AppSet_{HOLD}$ is released if the $AppSet_{next}$ (the next $AppSet$ enters the run queue) is the same as the $AppSet_{HOLD}$; b_{a112} , HV denotes the historical volume (number of non-idle CPU cycles) of a given $AppSet$ accumulated across a series of consecutive scheduling epochs. The $AppSet_{HOLD}$ is released if the time taken to process $AppSet_{HOLD}$ based on its HV average historical volume is larger than T_{remain} , which is the time from now until the predicted arrival of the next incoming $AppSet$ on hold; b_{a113} , the $AppSet_{HOLD}$ is released if the estimated completion time T_{comp} of the $AppSet_{HOLD}$ is expired.

The min duration constraint for the HOLD scheme defines the minimum scheduling duration or the initialization condition of the $AppSet_{HOLD}$ to avoid the situations where the amount of TIP saved by applying the HOLD scheme is less than the overhead generated by applying the scheme. There are three constraints under this category: b_{a121} , $Overhead_{HOLD}(AppSet_{HOLD})$ denotes the overhead generated by holding the $AppSet$. The minimum holding time of the $AppSet_{HOLD}$ must be long enough to reduce a number of CPU cycles CC_{saved} that is larger than the overhead generated; b_{a122} , $Overhead_{ADD}(AppSet_{HOLD})$ denotes the additional overhead inflicted by putting the $AppSet$ on HOLD for multiple cycles, where additional CPU idle cycles are generated due to the rearrangement of newly arrived $AppSet$ at the beginning of a new epoch. The additional overhead should be also covered by CC_{saved} ; b_{a123} , $Overhead_{FE}(AppSet_{HOLD})$ denotes the fringing effect overhead inflicted on other $AppSet$ by applying the HOLD scheme. This overhead includes power dissipated in keeping additional memory storage and generating additional TIP in the future due to unpredictable threads. The minimum holding time should suffice a CC_{saved} to cover the overhead.

The scheme duration constraint for the HOLD scheme defines the number of scheduling epochs that the $AppSet_{HOLD}$ can last while adhering to the max and min scheduling duration. There are three constraints under this category: b_{a131} , $AppSet_{HOLD}(TIP_{generate})$ denotes the TIP generated during the application of HOLD scheme, $AppSet_{HOLD}(TIP_{ave})$ denotes the average TIP dissipated by the $AppSet$ on hold historically. The $AppSet_{HOLD}$ will retire at its max scheme duration if its average TIP is always larger than its generated TIP; b_{a132} , TIP_{reduce} denotes the amount of TIP reduced in term of CPU cycles. It is calculated at the end of each epoch by accumulating the difference between the $TIP_{generate}$ and the TIP_{ave} . Once the $TIP_{generate}$ is noticed to be larger than TIP_{ave} , $AppSet_{HOLD}$ is then only allowed to retire at its max scheme duration if TIP_{reduce} is large enough to cover the additional TIP; b_{a133} , the HV of $AppSet_{HOLD}$ is compared to its average value $AppSet_{ave}$ at the end of each epoch. The $AppSet_{HOLD}$ will be terminated if HV at current scheduling epoch is larger than its average historical volume.

SUPER Scheme:

The max duration constraint for the SUPER scheme defines the maximum scheduling duration or the release condition of the $AppSet_{SUPER}$ ($AppSet$ under the SUPER scheme. It contains $AppSet$ with super slice $AppSet_{superSlice}$ and $AppSet$ with Small Slice $AppSet_{smallSlice}$. In an $AppSet_{SUPER}$, the $AppSet_{superSlice}$ is

given more time slices, while $AppSet_{smallSlice}$ is given less time slices.). There are three constraints under this category: $b_{\alpha 211}$, $AppSet_{superSlice}(HV)$ denotes the number of epochs required to process a super slice with an average historical volume HV . The SUPER scheme will retire if its $AppSet_{superSlice}(HV)$ requires more epochs than T_{remain} . This constraint prevents a super slice from hogging the run queue; $b_{\alpha 212}$, $AppSet_{next}(HV)$ denotes the historical average volume of the $AppSet$ at next scheduling epoch. $AppSet_{SUPER}(HV)$ denotes the historical average volume of the $AppSet$ at the current scheduling epoch applying SUPER scheme. The SUPER scheme will retire if the $AppSet$ volume at the next epoch is larger than the current epoch to avoid race condition between consecutive $AppSets$; $b_{\alpha 213}$, the SUPER scheme will retire if the estimated completion time T_{comp} for $AppSet_{smallSlice}$ is violated.

The min duration constraint for the SUPER scheme defines the minimum scheduling duration or the initialization condition of the $AppSet_{SUPER}$. There are three constraints under this category: $b_{\alpha 221}$, $Overhead_{preemptive}$ denotes the preemptive overhead generated during the event where application with higher priority interrupts the scheduling sequence of the run queue. Additional TIP is generated if it occurs in the middle of a super slice, hence it is predicted and checked while determining the min scheme duration. The number of SUPER scheme epochs is expected to generate enough CC_{saved} to cover the predicted preemptive overheads; $b_{\alpha 222}$, $Overhead_{reslice}$ indicates the time slice reslicing overhead of the SUPER scheme. It occurs at the beginning of each scheduling epoch under the SUPER scheme, where the member $AppSets$ of the $AppSet_{SUPER}$ get their time slice size adjusted to satisfy the SUPER scheme arrangement. The additional overhead generated should be covered by the CC_{saved} ; $b_{\alpha 223}$, $Overhead_{FE}(AppSet_{SUPER})$ denotes the fringing effect overhead caused by the SUPER scheme. It is a result of memory congestions caused by unpredicted $AppSet_{smallSlice}(HV)$ increment. The additional TIP caused by the fringing effect overhead are meant to be covered by the CC_{saved} by the SUPER scheme while determining the min scheme duration.

The scheme duration constraint for the SUPER scheme defines the number of scheduling epochs that the $AppSet_{SUPER}$ can last while adhering to the max and min scheduling duration. There are three constraints under this category: $b_{\alpha 231}$, $AppSet_{SUPER}(TIP_{generate})$ indicates the TIP generated by the member $AppSets$ of the $AppSet_{SUPER}$. The member $AppSet$ with the highest TIP generation is chosen to be given more time slices or the super slice. It is compared with the historical average TIP generated by the same $AppSet$ ($AppSet_{ave}(TIP)$). The SUPER scheme will retire when it reaches its max scheme duration or the TIP generated exceeds its historic average TIP; $b_{\alpha 232}$, the member $AppSet$ with higher HV is given the super slice and the member $AppSet$ less HV is given the small slice. The SUPER scheme will either rearrange the time slice or retire if the HV of $AppSet_{smallSlice}$ gets larger than $AppSet_{superSlice}$; $b_{\alpha 233}$, if it is predicted that the $AppSet_{next}$ in the run queue is the same as the $AppSet_{smallSlice}$, the super slice will be given to the next best candidature $AppSet$ based on $b_{\alpha 231}$ and $b_{\alpha 232}$ to avoid the violation of $AppSet_{smallSlice}(T_{comp})$.

DEDICATED Scheme:

The max duration constraint for the DEDICATED scheme defines the maximum scheduling duration or the release condition of the $AppSet_{DEDICATED}$ ($AppSet$ scheduled with the DEDICATED scheme). There are three

constraints under this category: $\mathbf{b}_{\alpha 311}$, $Core_{DEDICATED}(E_{budget})$ denotes the energy budget E_{budget} of a CPU core applying the DEDICATED scheme, which is the accumulated amount of energy saved by Re DFN scheduling schemes. $Core_{DEDICATED}(E_{consumed})$ denotes the energy consumed to increase the operating frequency of CPU core applying the DEDICATED scheme. The DEDICATED scheme will retire once the energy consumed exceeds the energy saved; $\mathbf{b}_{\alpha 312}$, $Core_{other}(TIP_{generate})$ indicates additional TIP generated in other CPU cores by housing $AppSets$ migrated from $Core_{DEDICATED}$. $Core_{DEDICATED}(TIP_{reduced})$ indicates the TIP reduced by applying the DEDICATED scheme in $Core_{DEDICATED}$. The DEDICATED scheme will retire if TIP generated additionally is larger than the TIP reduced; $\mathbf{b}_{\alpha 313}$, $AppSet_{DEDICATED}(HV)$ denotes the volume HV of the $AppSet_{DEDICATED}$ processed under the DEDICATED scheme. $AppSet_{ave}(HV)$ denotes the average historical volume of the same $AppSet$. The DEDICATED scheme will retire if the volume of the $AppSet_{DEDICATED}$ processed dominates its historical average.

The min duration constraint for the DEDICATED scheme defines the minimum scheduling duration or the initialization condition of the $AppSet_{DEDICATED}$. There are three constraints under this category: $\mathbf{b}_{\alpha 321}$, $Overhead_{migration}$ denotes the migration overhead generated by the DEDICATED scheme. It is a result of migrating the target $AppSet$ to the DEDICATED CPU core. The DEDICATED should be scheduled for more than a number of epochs to accumulate enough CC_{saved} to cover the overhead; $\mathbf{b}_{\alpha 322}$, $Overhead_{reservation}$ denotes the core reservation overhead generated by reserving the DEDICATED CPU core for the $AppSet_{DEDICATED}$. It increases $E_{consumed}$ on the other CPU cores to accommodate predictable future $AppSets$ originally planned for the DEDICATED core. The min scheme duration should last a number of epochs for DEDICATED scheme to accumulate enough CC_{saved} ; $\mathbf{b}_{\alpha 323}$, $Overhead_{FE}$ indicates the fringing effect overhead generated by additional $AppSets$ migrated from the DEDICATED core in other CPU cores. Additional TIP is inflicted as the prediction of $AppSet_{next}$ in other CPU cores less accurate. The min scheme duration will last a number of epochs to accumulate enough CC_{saved} .

The scheme duration constraint for the DEDICATED scheme defines the number of scheduling epochs that the $AppSet_{DEDICATED}$ can last while adhering to the max and min scheduling duration. There are three constraints under this category: $\mathbf{b}_{\alpha 331}$, $AppSet_{DEDICATED}(TIP_{generate})$ indicates the amount of TIP generated when the DEDICATED scheme is applied. $AppSet_{ave}(TIP)$ is the average TIP generated by the same $AppSet$ over time. TIP generated by the DEDICATED scheme is required to be lower than 70% of the average TIP, as the overhead composition of the DEDICATED scheme is more complicated comparing to the other two schemes. The DEDICATED scheme will retire prematurely if the constraint is breached; $\mathbf{b}_{\alpha 332}$, $AppSet_{DEDICATED}(Core_{other}(TIP))$ indicates the TIP generated by the $AppSet_{DEDICATED}$ while processing in other CPU cores. The DEDICATED scheme will retire before hitting its max scheme duration, if the TIP generated in other CPU cores is less than the TIP generated by the $AppSet_{DEDICATED}$ in the DEDICATED core; $\mathbf{b}_{\alpha 333}$, the DEDICATED scheme is retired if its migration overhead generated is larger than the TIP it can reduce.

Layer 6 Parameter Optimization

This layer conducts supervised training on the *AppSet* PMP predicted against the PMP monitored. The predicted PMP is derived based on scheduling scheme weightage and scheme parameter in Layers 4 and 5. The corresponding *AppSet Table* PMP values are used to predict the resulting PMP of the *AppSet* optimized by the Re DFN. And the accuracy of the prediction is hidden and difficult to be adjusted for optimization at runtime. Hence in order to testify and improve the result *AppSet* optimization, the predicted values of α , $\mathcal{P}_\alpha = \{p_1, p_2, \dots, p_n\}$, where $p_n\{p_n^1, p_n^2, \dots, p_n^t\}$ is the predicted PMP vector retrieved from *AppSet Table*, n denotes the column of PMP entry and t denotes the row. The monitored PMP values $\mathcal{Q}_n = \{q_1, q_2, \dots, q_n\}$ has the same dimension as that of $\mathcal{P}_\alpha$. The learning is realized with a multilayer perceptron (MLP) as shown in Figure 5.18. Table 5.7 lists the hyperparameters of Layer 6.

Figure 5.18 detailed layout of layer 6

Table 5.7 Hyperparameters of Layer 6

	Number of Neurons	Activation Function	Weight Matrix Size	Input Size
Input Layer $\mathcal{A}$	25	Sigmoid	30×25	10×25
Hidden Layer $\mathcal{B}$	30	Sigmoid		Output Size
Output Layer $\mathcal{C}$	25	Sigmoid	25×30	10×25

The MLP is randomly initialized. The hyper parameters are set using cross validation and the neural network is regularized with L2 regularization to avoid over fitting. At the forward propagation phase $\mathcal{P}_\alpha$ is passed into the MLP. The values at the output layer will be compared with the label $\mathcal{Q}_n$. And the MLP gets trained in backpropagation by passing the error term δ backwards from the output layer to the input layer. The error term is extracted every learning cycle to make corrections on the *AppSet Table* PMP values. Each PMP entry will be updated as a hidden layer bias term with $\hat{b} = b - \eta[\frac{1}{t}\delta]$. The MLP is aimed to be converged at the end of the training and the error term gets continuously update every learning cycle, hence the PMP values in the *AppSet Table* get an online optimization towards describing the PMP of the targeting *AppSet* accurately and enables the Re DFN to dynamically adjust the scheduling scheme at runtime.

5.4 Re DFN optimization

To reduce computation overhead and energy consumed in the process of Neural Network computation, Neural Network compression techniques such as layer-wise sensitivity analysis, neural network pruning, matrix sparsity reduction, weight quantization and sharing were implemented.

5.4.1 Sensitivity Analysis and Neural Network Pruning

Pruning a Neural Network will reduce the computation complexity and memory size occupied by it at runtime. Prior to the pruning process, a sensitive analysis has to be conducted in between neurons and the weights connecting them. The sensitivity analysis will serve as a basis for the Neural Network to prune weights and neurons with less significance in modeling the current problem learnt.

Figure 5.19 shows an example of Layer 2 in the proposed DNN pruned into a simply neural network after the layer-wise sensitivity analysis. The hidden neurons and output variables of the sub layers are reduced, and their relevant weights are pruned. In this example the Neural Network size has been reduced to 22 weights, that is only one fifth of its original size of 105 weights before pruning. The idea of sensitivity analysis is first brought into the field of neural network by Garson in 1992[135] and Goh in 1995 [136]. The aim of the technique was to identify the relative importance of explanatory variables from the response of their resulting variables in a neural network by decoupling their connecting weights. The term explanatory variables are interchangeably adopted as independent variables, where in the context of Neural Network it is normally used to address the input variable which is independent and free from influences of other variables. In the early hypothesis of sensitivity analysis, it was assumed that in a particular neural network, all connecting weights from specified input nodes that propagate through various layers of hidden layers to the specific output node of interest are identifiable through a process of mathematical computation. By repeating this process for all explanatory variables, in this case all specified input node variables of interest, a list of summed weights between every input variable that is related and resulted to every output variable is generated at the end of the computation. After rearranging the connecting weights according to the list, a single valued weight parameter is computed for each explanatory variables to mark its significance to other variables in the neural network. The study of the Goh algorithm used to compute the significance of explanatory variables indicates a significance value as an absolute magnitude with no sign preserved.

Figure 5.19 Simplified Layer 3 of the Re DFN before and after pruning

To implement the algorithm in the proposed DNN Scheduler, the analysis is carried out in a layer-wise manner, where each layer will have a separated sensitivity analysis to address the significance of variables in each layer without causing over pruning the network. As shown in Figure 5.20, there are two pairs of matrices generated for the 3-layer Performance TIP generalization sub layer in Layer 2. While Weight Matrix W_l contains all connecting weights between Layer L_l (14 neurons) to L_{l+1} (5 neurons), where $l+1 = m$, the other matrix needed for conducting the sensitivity analysis from Layer L_l to L_m is the Diagonal Matrix, $V'_m = \text{diag}(v'_{1m}, v'_{2m}, \dots, v'_{tm})$, where v'_{tm} is the derivative of an output neuron v_{tm} in Layer L_m . The other sensitivity analysis matrix pair is the Weight Matrix W_m which indicates the connecting weights between Layer L_m (5 neurons) to L_n (7 neurons), where $n = m + 1$; and the Diagonal Matrix V'_n which is used to conduct sensitivity analysis between explanatory variables of Layer L_m and responsive variables in L_n .

Figure 5.20 Weight Matrix in Sub Layers of Re DFN Layer 3

As shown in Figure 5.21, the sensitivity matrix computation pair is computed with Matrix multiplication to derive the Sensitivity Matrix S_{lm}^e for Layer L_l to L_m and S_{mn}^e for Layer L_m to L_n , where the first subscript from the left indicates the upper layer and the second subscript indicates the lower layer, and e indicates the batch number of the training set. Once the Sensitivity Matrix is computed, each and every member value in the matrix represents the sensitivity level of the influence of the explanatory variable of the upper layer over the responsive variable of the lower layer. In the process of pruning, the matrix is checked against a threshold value to determine whether the targeted weight should be pruned or kept.

Figure 5.21 Sensitivity Matrix Computation in Sub Layers of Re DFN Layer 3

The threshold is named as the pruning significance, it is to be determined before carrying out detailed pruning operations. As shown in Figure 5.22, the pruning significance of connecting weights between Layer L_l and L_m is vector $\Phi_{c,L_l} = \{\varphi_{o,L_m}, \varphi_{o+1,L_m}, \dots, \varphi_{o+13,L_m}\}$, where φ_{o,L_m} denotes the matrix coordinate of the most significant responsive variable from Layer L_m for the given explanatory variable from Layer L_l as $\varphi_{o,L_m} = \max_i \{S_{io,lm}^e\}$. Besides the pruning significance, there is another bias factor used to optimize the degree of pruning, the pruning factor F , while $\frac{S_{io,lm}^e}{\varphi_{o,L_m}} > F$. The Sensitivity Matrix value at given coordinate has to be

Each layer of the Re DFN is processed individually for Weight Sharing and the quantized weights are only shared among the Weight Matrices of their dedicated Layer. As Shown in Figure 5.23, the Weight Matrix W_l is processed to group its connecting weight into four groups. Each different color stands for a different weight group. There are h connecting weights in the Weight Matrix and as for Layer L_l , $W_y = \{w_{11,L_y}, w_{12,L_y}, \dots, w_{ij,L_y}\}$, $i = 5, j = 14$ for $y = l$ and $h_l = i \times j = 70$. The connecting weights are grouped via k means clustering algorithm into g clusters with centroids of $C_y = \{c_{1,L_y}, c_{2,L_y}, \dots, c_{g,L_y}\}$, $g = 4$ for $y = l$. Once the centroids are initialized, the connecting weights are clustered accordingly by satisfying the cost function of $(c_{g,L_y}) = \sum_{w_{ij,L_y} \in c_{g,L_y}} \|w_{ij,L_y} - \mu_{g,L_y}\|^2$, where μ_{g,L_y} is the mean of the centroid c_{g,L_y} .

Figure 5.23 Weight Matrix W_l before and after Clustering based Weight Grouping

As shown in Figure 5.24, once the weight grouping is completed, the values from the Sensitivity Matrix S_{lm}^e of W_l are also assigned with the same group of their corresponding Weight Matrix counterpart. Each member value of the Sensitivity Matrix $S_{lm}^e = \{s_{11,L_{lm}}^e, s_{12,L_{lm}}^e, \dots, s_{ij,L_{lm}}^e\}$, $i = 5, j = 14$ is grouped according to the corresponding weight clusters, for each Sensitivity Matrix value $s_{ij,L_{lm}}^e \in c_{g,L_l}$, while $w_{ij,L_l} \in c_{g,L_l}$.

Sensitivity Matrix S_{lm}^e , Highlighted according to the weight grouping of W_l

-0.10	0.09	-0.02	0.01	0.02	0.09	0.01	0.01	0.01	0.01	0.14	0.11	-0.09	0.05
0.06	-0.03	0.00	-0.01	-0.02	-0.03	-0.09	0.13	0.04	0.04	-0.09	-0.08	0.03	0.01
0.05	0.01	-0.02	-0.08	0.01	-0.05	-0.15	-0.01	-0.13	0.03	-0.05	0.05	0.01	0.05
-0.07	0.02	0.02	-0.18	0.07	-0.04	-0.04	-0.02	0.09	-0.07	-0.02	-0.07	0.02	0.00
0.13	-0.07	-0.06	0.05	-0.13	0.09	0.02	0.03	0.01	-0.08	-0.02	-0.19	0.11	0.05

 Figure 5.24 Sensitivity Matrix S_{lm}^e grouped according to weight groups in Clustered W_l

As shown in Figure 5.25, the Sensitivity Matrix values in each cluster are used to compute the Regularization factor $R_y = \{r_{1,L_y}, r_{2,L_y}, \dots, r_{g,L_y}\}$, where r_{g,L_y} is the mean of the Sensitivity Values belongs to the same cluster and $g = 4$. The Regularization factors are added to the centroid of their corresponding clusters to form the Regularized Centroids. The Regularized Centroids are used to serve as the shared weight of the corresponding layer.

	C_l		R_{lm}		Regularized Centroids
c_{4,L_l}	2		-0.16	$r_{4,L_{lm}}$	1.84
c_{3,L_l}	1.5		-0.35	$r_{3,L_{lm}}$	1.15
c_{2,L_l}	1		-0.1	$r_{2,L_{lm}}$	0.9
c_{1,L_l}	0		0.32	$r_{1,L_{lm}}$	0.32

$\oplus =$

Figure 5.25 Centroid Regularization with Sensitivity Constraints

5.4.3 Sparse Matrix Storage Format

A fully connected DNN is always memory hungry as there are a vast amount of matrices to be computed at the training phase. Such matrix computation complexity very much limits the usage of large scaled private Neural Network to mobile computing platforms [141] [142]. There are two aspects in countering this limitation: 1. removing sparsity in matrices with formatting to utilize fewer bits per entry and save on storage memory; 2. reducing the size of the matrices with different storage format to fit dense matrices into cache for better CPU throughput and generation of TIP related power dissipation problems. To solve the problem a layer-wise Sparse Matrix Indexing has been adopted, to ensure high efficiency in matrix operation of the Re DFN.

Figure 5.26 demonstrates Sparse Matrix Compression of Weight Matrix W_l with two types of Matrix Storage Format, the Diagonal (DIA) [143] and the Compressed Sparse Row (CSR) [144]. The DIA format stores a sparse matrix by looking for Diagonals that contains non-zero values in the original matrix. Each of the Diagonals $d_c = \{w_{ij,L}, w_{i+1,j+1,L}, \dots, w_{i+f,j+f,L}\}^T$, where f = number of rows in the original Matrix and c is the column number of Diagonal Value Matrix D . The Diagonals constructed with a searching within the original matrix by modifying the value of i, j and f . The search starts at $i=f, j=1$ with f steps in the search of one vector d_c .

For each non-existing matrix coordinate the entry will be pad with an asterisk sign. Upon completion of a search if the vector $d_c \neq \vec{0}$, it will be listed in Matrix $D = \{d_1, d_2, \dots, d_z\}$, where z is the number of non-zero d_c vectors. The original Weight Matrix position of diagonal vectors are denoted with the Diagonal Distance Offset, where the position of the first vector in D is the origin. The difference of the j value between two neighboring non-zero vectors is calculated as an offset and is added to the Diagonal Distance Offset vector for the corresponding diagonal vector.

The CSR format remedies the matrix sparsity by searching for non-zero values in the original matrix and form the Value Vector $V = \{v_1, v_2, \dots, v_p\}$, where p is the number of non-zero entries in W_l . The column of each noted non-zero value in the original Weight Matrix is recorded in the Column Index Offset Vector $J = \{i_1, i_2, \dots, i_p\}$, where all i_p of the first v_p in a row equals to $j-1$ and j is the column coordinate of the $w_{ij,L}$ for the corresponding v_p . All other i_p in the same row equals to $j-1-i_{p-1}$. A Row Offset is added to the Row Offset Vector $\mathcal{R} = \{r_0, r_1, \dots, r_q\}$, where q is the number of rows of the original matrix, r_q is the sum of non-zero entries in the current and all previous rows, and r_0 is the origin term with a value of zero. Upon completion of the compression original matrix can be reconstructed by determining the row of each value in the Value Vector with the Row Offset Vector and their column with the Column Index Offset Vector.

Diagonal Storage Format

Diagonal Values

*	1.5	0	1.5	1.5	2	2	0	0	2	0
*	1	0	1	1.5	0	1.5	0	1.5	1	0
*	0	0	1	1	2	1	0	0	0	1
0	2	1.5	1	0	0	0	0	1	0	*
1.5	1.5	0	1	2	2	1.5	1.5	2	*	*

Diagonal Distance Offset

1	3	4	2	2	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---

Weight Matrix W_l with shared weights

1.5	0	0	1.5	0	1.5	2	2	0	0	2	0	0	0
0	1	0	0	1	0	1.5	0	1.5	0	1.5	1	0	0
0	0	0	0	0	1	0	1	2	1	0	0	0	1
0	0	0	2	1.5	0	1	0	0	0	0	0	1	0
0	1.5	0	0	1.5	0	0	1	0	2	2	1.5	1.5	2

Diagonal Distance Offset = $4 - 1 = 3$

Row Offset of Row 3 = $6 + 6 = 12$

Compressed Sparse Column Storage Format

Column Index Offsets

0	3	2	1	1	3	1	3	2	2	2	1	5	2	1	1	4	3	1	2	6	1	3	3	2	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Values

1.5	1.5	1.5	2	2	2	1	1	1.5	1.5	1.5	1	1	1	2	1	1	2	1.5	1	1	1.5	1.5	1	2	2	1.5	1.5	2
-----	-----	-----	---	---	---	---	---	-----	-----	-----	---	---	---	---	---	---	---	-----	---	---	-----	-----	---	---	---	-----	-----	---

0	6	12	17	21	29
---	---	----	----	----	----

Row Offset

Figure 5.26 Sparse Matrix Storage Format Comparison on Weight Matrix W_l

As shown in Figure 5.26, there are 55 diagonal values and 11 diagonal offset values used to represent the weight matrix in Diagonal Storage Format, and each value is represented with 2 bits. There are 29 values each represented with 2 bits, 29 index offset values each represented in 3 bits and 6 row offset values each represented in 5 bits are used to represent the weight matrix using Compressed Sparse Column (CSC) Format. While there are 90 values in the original uncompressed weight matrix and each is represented with 32 bits. Hence the total number of bits used to store the weight matrix shown in Figure 5.26 are: $(55 + 11) \times 2 = 132$ bits in Diagonal Storage Format, $29 \times 2 + 29 \times 3 + 6 \times 5 = 175$ bits in CSC format and $90 \times 32 = 2880$ bits in the original weight matrix. Comparing the total number of bits used to store the weight matrix in the original uncompressed format and CSC format, the storage space is reduced by 94%. The Diagonal Storage Format is

utilized when the number of diagonals with non-zero values in a matrix is less than the number of columns, where the CSC is utilized when the number of non-zero values when the Sparse Matrix is much more structured with multiples of zero valued diagonals. The indexing process is carried out on a layer-wise manner, where each Weight Matrix is processed separately and the type of storage format has been dynamically chosen based on the sparsity of the Weight Matrix for optimized performance.

5.5 Results

The proposed Re DFN is constructed using TensorFlow [145] with RBM extension, the performance is tested in experiments against three sets of training Set. The training sets are simulated with GEM5 [96] and having their output performance traces recorded for the training of Re DFN. Offline training is then conducted to generate *AppSet Table*, Power Dissipation Problem table, scheduling scheme decisions and parameters. The generated tables and parameters are used to derive runtime scheduling decisions online to optimize power dissipation. Each experiment consists of a performance trace of 1300 seconds with a sampling rate of 1 sample per second which is equivalent to 130 training sets for Re DFN with a mini batch size of 10 samples. The number of training set is determined based on a trial and error approach. It is the most time efficient to produce meaningful result with 130 training sets in one experiment. The performance trace reflects a series of simulated Android apps executed randomly in a sequential manner. There are three different training sets representing different app usage scenarios: Training Set 1 consists performance trace of a randomly picked individual apps executed in sequence; Training Set 2 constructed with performance trace of 2 randomly picked apps executing in pairs; and Training Set 3 contains 3 randomly pick apps executing simultaneously. After offline training phased the Re DFN is then tested against 3 test sets of with the same number of samples but different app sequence as comparing to the training sets.

The tested Re DFN is then deployed in the environment simulated by GEM5 simulator and McPAT [97] to provide accurate power dissipation of CPU cores under different thermal configuration. The simulated environment reconstructed a three 28 nm cluster architecture with two cores each: Cortex A57 (out-of-order) with a maximum operating frequency of 1.9 GHz, A53 (in-order) with maximum frequency at 1.3 GHz and A35 (in-order) with maximum frequency at 1 GHz. apps in the training sets and test sets are generated with Moby Bench [102] and the Linux Schedulers for comparison are the GTS under Linux Kernel 3.10 and the IKS under Linux Kernel 3.14.

5.5.1 Training Error and Test Error

To have a detailed breaking down for more insightful information, the Re DFN (Deep Function Network) is trained in a layer by layer manner with different training sets and tested against the test sets. During the training a mini batch is injected to Layer 1, then after learning (Contrast Divergence k for Layer 1 and Layer 4, backpropagation for other layers) the resulting Layer 1 output gets forward propagated to the following layers and the same feeding forward process repeats for the rest of the network. During the learning phase of a mini

batch the means square error (MSE) derived in updating the neural network parameters is recorded and plotted against the accumulative learning epoch per training set per layer. After learning the 3 training sets the Test Sets is injected to the Re DFN for result evaluation and similarly the MSE is recorded and plotted against the learning epoch. Table 5.8 shows the average training state and testing state error of each layer. The average error is calculated by normalizing the mean square error from each training set. The average training error and testing error has a rather large deviation in terms of percentage, for example the average testing error is 43% larger compare to the training error. In some neural network applications this could mean over fitting during cross validation and should be solved with regularization or drop out technique. However one of the aims of implementing Neural Network in the Re Scheduler is to remedy its Performance Metric Parameter prediction error, which is averagely 6.6% across all PMPs. By comparing the average testing state error from all layers, implementing the Re DFN reduces the prediction error rate from 6.6% to 2.53%, which is a 61% improvement. Translating prediction accuracy improvement to power consumption, there is a 2% of improvement in the overall power reduction. Hence the difference between the training and testing state error of less than 1% is not much a concern in the context of the Re DFN.

Table 5.8 Hyperparameters of Re DFN and the Average Error Rate

Layer	No. of Input Neurons	No. of Hidden Neurons	No. of Output Neurons	Training State Error	Testing State Error
1	25	50	135	1.90%	2.20%
2	100	150	75	2.10%	2.66%
3	275	284	275	1.74%	2.40%
4	9	3	3	1.74%	1.91%
5	27	27	9	2.25%	2.91%
6	25	30	25	2.18%	3.12%

Figure 5.27 shows the detailed training and testing error over Training Set 1 to Training Set 3. It is shown that the best least training error and testing error is in Training Set 3 due to the fact that the DFN is constructed to model the problem of Thread Interference Power (TIP) exists in the concurrently executed apps. The more apps are there in the targeted *AppSet* the better the performance in the TIP reduction, where Training Set 3 consists of 3 apps. That explains why Training Set 1 gets the most training and testing error. However Training Set 2 has the best result in terms of training error and testing error convergence. This is because in order to limit the size of the Neural Network for optimized performance in a mobile platform the Re DFN is constructed to model problems up to 3 apps in an *AppSet*. Hence the number of neurons are customized to receive and process inputs of an *AppSet* consisting of 3 apps. By having an input layer with number of neurons that is exactly matching to the number of inputs is more likely to result in overfitting. The slightly larger gap between training error and testing error in layer 1 Training Set 3 is one example of overfitting.

Figure 5.27 Layer 1 Training Error and Testing Error

Figure 5.28 shows the detailed training and testing error breaking up according to Training Sets in Layer 2. Similar to Layer 1, the best training and testing error convergence exists in Training Set 2 due to the overfitting problem. However the least training and testing error is not occurred in Training Set 3 in Layer 3, which is due to the reason that there is a Recurrence Neural Network (RNN) resides in Layer 2. The function of the RNN is to predict the incoming sequence of apps by accessing the current app sequence and it is normally more difficult to train than other types of Neural Networks. Hence the entire Layer 2 gets more training and testing errors as comparing to Layer 1. In addition, the Training Set 3 becomes throttled at the RNN due to the complex incoming pattern of an *AppSet* containing 3 apps and resulted in more training and testing error as comparing with Training Set 2. For Training Set 1, the RNN would be not only expecting *AppSet* with 1 app due to the structure of the Neural Network, hence the predictions errors are even higher than Training Set 3.

Figure 5.28 Layer 2 Training Error and Testing Error

Figure 5.29 shows the detailed training and testing error break up of Layer 3. Generally the training and testing errors are not as good as comparing to the previous 2 layers across all training sets. That is due to the problem that Layers 1 and 2 break each sample in training sample into 10 inputs and inject 1 input per learning cycle. On the other side Layer 3 combines all 10 inputs into sample and learning in 1 epoch. Hence this causes Layer 3 to have less learning epoch in total and more unresolved training and testing errors. However the overall error rate is still promising and viable to be used for Power Dissipation Problem identification as the accuracy is maintained not only with the prediction from the Autoencoder network, but also the bias terms deduced in the PDP table.

Figure 5.29 Layer 3 Training Error and Testing Error

Figure 5.30 shows the detailed training and testing error break up of Layer 4. The function of Layer 4 is to determine the scheduling scheme based on the Power Dissipation Problems (PDPs) identified. The layer is constructed with the Restricted Boltzmann Machine (RBM) to generate a weightage for each scheduling scheme. As RBMs are input size sensitive, performance degradation is likely to happen in the case that the difference between the hidden and visible layer is large. The output of Layer 3 denotes the PDP identified to serve as the input of Layer 4 and it is likely to have more variation of different PDP in an *AppSet* with more apps. Hence comparing to Training Set 1, Training Set 2 contains more PDP identified; comparing to Training Set 3, Training Set 2 contains less PDP identified. The number of PDPs identified in Training Set 2 matches the visible layer of Layer 4 the best. The unmatched input size with the number of neurons in visible layer has caused the learning accuracy impairment in the RBM. However even with the degradation in training and testing error, the overall accuracy is still acceptable for scheduling scheme prediction.

Figure 5.30 Layer 4 Training Error and Testing Error

Figure 5.31 shows the detailed training and testing error break up of Layer 5. Layer 5 determines the scheduling scheme parameters with Autoencoders masked with the conditional bias term of each scheduling scheme. It is shown that in the case of *AppSet* with 1 app in Training Set 1 there are limited numbers of PDPs identified, hence the chances of missing one or two scheduling scheme inputs from layer 4 is high. By having missing inputs in the input layer Training set 1 has a rather sparse hidden layer for computation, resulted in high training error and less convergence between the testing error. The accuracy result is much better for Training Sets 2 and 3, and Training Set 3 has the least training and testing error with a rather small difference comparing to Training Set 2. That is due to the fact that the scheduling scheme generated in Layer 4 is in probability form, hence all input neurons in the input layer will always get injected with values for Training Sets 2 and 3.

Figure 5.31 Layer 5 Training Error and Testing Error

Figure 5.32 shows the detailed training and testing error break up of Layer 6. The function of Layer 6 is to determine the resulted Thread Interference Power (TIP) in Performance Metric Parameters (PMPs) after optimization from the Re DFN Scheduling Schemes and improve the scheduling optimization process based on the actual monitored PMP. This layer has the most average training and testing error overall, which is a result of being the last layer in the entire DFN and the error terms reside in the output of the previous layers gets propagated through. Training Set 3 gets the least training and testing error generated out of all three training sets, which proves the assumption that the TIP reduction method used in Re DFN works better in an environment with multiple apps executing in parallel.

Figure 5.32 Layer 6 Training Error and Testing Error

5.5.2 Size Reduction and Accuracy after Optimization

To fit the Re DFN into the memory, cache and power budget of a mobile platform, various approaches have been adopted to reduce the size and the computation overhead of the network. As discussed in Section 5.4 to reduce the size and computation complexity of the neural network, weight and neuron pruning has been applied. However a pruned neural network will surely be exposed to side effect such as losing its accuracy. Hence a

prior study of the relationship between the prune factor F and the inversed testing error and test accuracy has been constructed, where the three training set have been injected against different layers to deduce the testing error, after each of the layers being pruned with a different F . The larger the prune factor the more connecting weights and neurons in a neural network can be pruned. The result is shown in Figure 5.33, where Layer 1 is able to take a prune factor up to 2 without significant accuracy degradation but Layer 2 is only able to be pruned by a prune factor of 1. The most pruning proof layers are Layers 3 and 6.

Figure 5.33 Prune Factor Plot against Test Accuracy

The largest prune factor applicable to the respective layer is then applied after conducting the accuracy test. Besides neural network pruning, weight sharing and matrix compression has also being applied to the entire Re DFN. As shown in Table 5.9, the original number of bits to represent an entry in a weight matrix is a 32-bit float number. After weight sharing the average numbers of bits are reduced to 8, which is equivalent to a $\frac{32-8}{32} = 75\%$ reduction in the neural network size even with the additional Matrix Compression Index terms (averagely 8 bits each) added. After pruning the respective layers, their No. of Weights are less than the original by the Weights Pruned. The compression percentage of the pruned neural network comparing to the original neural network is the Overall Optimization, which can be computed with Equation (5.16),

$$\frac{(No. of Weights \times (1 - Weights Pruned) + Matrix Compression Index) \times Shared Weight bits}{No. of Weights \times Original Weight bits} \quad (5.16)$$

Table 5.9 Neural Network Compression Percentage after Optimization

Layer	No. of Weights	Weights Pruned	Original Weight bits	Shared Weight bits	Matrix Compression Index	Overall Optimization
1	500	26%	32	8	10	19%
2	3200	13%	32	8	40	22%
3	3200	47%	32	8	30	13.5%
4	27	11%	32	8	6	27.8%
5	108	31.5%	32	8	20	21%
6	1500	51%	32	8	60	13.25%

Figure 5.34 shows the scheduler computation overhead reduction of the pruned Re DFN over the original Re DFN in terms of cache miss, TLB miss, Branch miss and migrations. By optimizing the size of the Re DFN the matrix computation fits better in the cache units and inflicted fewer miss events, where the reduction is at least 50.23% of LLC (Last Level Cache or Level 2) load miss in layer 4 and can be as high as 83.81% of L1 data cache store miss in Layer 6. Due to the compressed size of the neural network, it is easier to be scheduled in parallel, hence there are fewer migration or context switch events, where the CPU migration reduction is at least 64.22% in layer 4, at most 81.65% in Layer 6. On top of that smaller sized memory blocks are used to store the matrices and resulted in branch misses reduction, where branch miss reduces at least 55.86% in Layer 4 and at most 73.76% in layer 1. By optimizing the size of the Neural Networks, makes keeping the Re DFN and running in a mobile platform more efficiently.

Figure 5.34 PMP Overhead Reduction in individual layers after Optimization

5.5.3 Power Reduction

Besides reducing the prediction error, there is another aim of reconstructing Re Scheduler with neural network, to reduce the overhead generated by the Scheduler itself. As shown in Figure 5.35 the overhead generated by Re DFN is much lower comparing to the Re Scheduler. The overhead is less than 6% and can be as low as 0.22% as compared to over 9% and 1.83% of the Re Scheduler. The reason is that unlike the Re Scheduler, in Re DFN the prediction, association and scheduling determination tasks are migrated to CPU idle time for semi Offline Learning, where less or no apps is processed concurrently. Hence the TIP generated is minimized. All the required parameters for scheduling decision are pre-calculated and collected in tables as a dynamic computing approach.

Figure 5.35 Comparison of Overhead generated of Re Scheduler and Re DFN

Figure 5.36 shows an overview of the performance metrics monitored at runtime using trained and pruned Re DFN against the Re Scheduler. The result is generated with the same test cases used in Chapter 4. Comparing to the original Re Scheduler in both kernel 3.10 and 3.14, the Re DFN gains average reduction from most of the miss related PMP events across all test sets from both kernels. The average reductions are calculated using the following equation, $\frac{1}{o \times m} \times \sum_i \sum_t \frac{(PMP_{i,t}(ReSCH) - PMP_{i,t}(ReDFN))}{PMP_{i,t}(ReSCH)}$, where $PMP_{i,t}(ReSCH)$ is the Re Scheduler PMP value in Figure 5.36, $PMP_{i,t}(ReDFN)$ is the PMP value of Re DFN, i indicates the PMP event (LLC pref miss, TLB load miss, etc.), t indicates the test set cluster combo (1 App S, 2 App B, etc.), $o = \{1, 2, \dots, 15\}$ is the number of PMP events considered in the calculation and $m = 9$ is the number of test set cluster combo considered in the calculation. Different PMP average reductions are calculated by considering different i and o . In Figure 5.36 four average reductions are calculated based on the PMP event range enclosed with rectangles of different colors. Blue, green, yellow and red rectangles indicate the PMP events considered to compute the average reduction of cache miss related, TLB miss related, branch miss and CPU utilization PMP from Re DFN and Re Scheduler respectively.

Figure 5.36 CPU on time observed in PMP values for Re DFN in Linux 3.14 and 3.10 kernel

There is an average reduction of 18.9% when comparing the red shaded Re DFN cache miss related PMP to the blue shaded Re Scheduler counterpart. There is an average TLB miss related PMP reduction of 14.6% by comparing the purple shaded Re DFN and the green shaded Re Scheduler PMPs. There is an average 17.4% branch miss PMP improvement by comparing the brown shaded Re DFN values and the Re Scheduler values shaded in gray. However the red shaded Re DFN CPU utilization PMP increased for by an average of 27.3% comparing to the yellow shaded Re Scheduler values, regardless of the reduction of CPU idle time achieved by the Re DFN. This is a result of the semi Offline learning approach of the Re DFN, making use of the idled CPU time to run learning tasks at vacant cores with low power budget. Hence, it reduced the overall Thread

Interference Power as parallel execution are avoided as much as possible, yet less power dissipation is generated even with a high CPU utilization time as the power budget allocated to the cores for learning tasks are low.

Table 5.10 Shows the overall power dissipation reduction of Re DFN over Re Scheduler, the Native GTS and IKS Scheduler. While Re DFN performs much better than GTS and IKS with average improvement of 22.57% and 10.6% respectively, there are not very much improvement in terms of power dissipation reduction by implementing Re DFN as comparing to what the original Re Scheduler has achieved. In the test against the Re Scheduler, Re DFN reduce only 5.77% of power dissipation in Linux kernel 3.10 and 3.6% of power dissipation reduction in kernel 3.14. Hence the Re DFN performs better than the Re Scheduler by 4.69% on average. The Re DFN has a small improvement over the Re Scheduler, because it follows the same power managing approach and aims to reduce scheduling overhead of the Re Scheduler. The improvement comes from reducing the scheduling overhead of the Re Scheduler using the neural network structure. The improvement complies with the results shown in Figure 5.35, where the Re DFN generates an average of 6% less of computation overhead comparing to the Re Scheduler. The improvement implies that the power dissipation reduction resulted from scheduling overhead reduction is quite significant, as 6% of overhead reduction reduces the power dissipation by 4.69%.

Table 5.10 Overall Power Consumption Result

Test Set	GTS (W)	Re Scheduler (W)	Re DFN (W)	Power Reduced Over Re Scheduler	Power Reduced Over GTS		IKS (W)	Re Scheduler (W)	Re DFN (W)	Power Reduced Over Re Scheduler	Power Reduced Over IKS
1 App	1.72	1.489	1.37	8.1%	20.5%		1.57	1.43	1.41	1.6%	10.3%
2 Apps	2.2	1.816	1.72	5.3%	21.8%		1.87	1.78	1.66	6.8%	11.3%
3 Apps	3.92	3.045	2.92	3.9%	25.4%		3.37	3.1	3.03	2.4%	10.2%
Average				5.77%	22.57%					3.6%	10.6%

As being promised by the developing technologies, executing the Re DFN on neural network friendly mobile hardware, the CPU utilization in Figure 5.36 can be much lower comparing to what has been achieved in the result. Thus a much better power dissipation reduction figure can be achieved. The implementation of the Re DFN enables automated tuning of the Re Scheduler on a new system and as technology advances, neural network is able to provide more reduction using dedicated AI chips on mobile device.

5.6 Conclusion

The raising interest in artificial intelligence has granted an uncharted field of solving dark silicon effect resulted problems in mobile system with an unmanned, automatic and yet adaptive approach. At the same time, power dissipation reduction should not be only focused on the applications being managed but as well as the application that is used to manage. Hence the development of a self-instructed efficient power manager is the core aim of this study. The Re DFN is designed to comprehend several shortcomings of the Re Scheduler: 1. Automate the Re Scheduler's system dependent initialization process using neural network based approach; 2. Reduce the scheduling overhead of the Re Scheduler by solving the complex decision making with the universal approximation capability of neural network; 3. Increase application pattern prediction using the time series data analysis capability of neural network. To address the shortcomings, the Re Scheduler is separated into different functions and the Re DFN realizes each function using neural networks as solving an approximation problem.

All neural network algorithms mentioned in this chapter are utilized in the Re DFN to map a partial function of the Re Scheduler. The Re Dynamic Function Network (DFN) uses the non-parametric universal approximation capability of neural network to resolve the prediction problems of the Re Scheduler efficiently and accurately. Various neural network structures, such as the Restricted Boltzmann Machine (RBM), Autoencoder and Recurrent Neural Network (RNN) etc. have been studied in order to determine the suitable design of Re DFN. Based on the neural network structures studied, the Re DFN has been constructed as a multi-layer neural network that consists of functions of multiple neural network structures studied.

The Re DFN has 6 layers and each layer reconstructs part of the functions of the previous proposed Re Scheduler: Layer 1 uses deep belief network to extract features from the Performance Metric Parameters (PMPs) monitored during system runtime; Layer 2 uses autoencoder and recurrent neural network to select features to reduce the feature space of PMPs monitored and the complexity of computation of trend prediction; Layer 3 uses stacked autoencoder to map PMP features selected with Power Dissipation Problems (PDPs) defined previously in the Re Scheduler, where table based dynamic computing approach is utilized to reduce the complexity of the PMP to PDP mapping; Layer 4 uses restricted Boltzmann machine to map PDPs identified with the scheduling schemes picked, that includes the HOLD, SUPER and DEDICATED scheme as proposed in Re Scheduler; Layer 5 uses autoencoder to set the parameters for each scheduling scheme picked with the mapping bias terms; Layer 6 uses multilayered perceptron to compare the PMP value predicted with the scheduling decisions made against the monitored PMP value at the end of the scheduling epoch. The difference between the monitored PMP value and the predicted PMP values is used to update the Re DFN to improve its accuracy.

However the computation overhead generated by the Re DFN was significant before optimization when compared to the original Re Scheduler. There were 4.57% of LLC load miss, 4.94% of TLB prefetch miss and 9.66% of L1 cache store miss generated by Re DFN without optimization, the values can be computed with dividing the optimized Re DFN values in Figure 5.35 by the difference between 1 and their respective average overhead reduction in Figure 5.34. There were 2.02% of LLC load miss, 4.57% of TLB prefetch miss and 3.05% L1 cache store miss generated by the Re Scheduler as shown in Figure 5.35. Hence various neural network optimization methods are applied. In the neural network pruning, each neuron in a network is examined and pruned according to their significance to the output. With the weight sharing, weights of different neurons are shared according to their level of identicalness and reduces overhead involved in computation. By using the matrix compression, the weight matrix of each neural network is compressed with specified matrix format, such as the Compressed Storage Column (CSC). After the optimization the size of the neural network is reduced by 80.5% (computed using Equation (5.16) and values in Table 5.9) and the overhead reduction has also been managed better comparing to the original Re Scheduler. The computation overhead caused by the optimized Re DFN had dropped to 1.29% of LLC load miss, 1.72% of TLB prefetch miss and 2.98% of L1 cache store miss respectively as shown in Figure 5.35. Comparing to the Re Scheduler, the Re DFN has gained improvement of 5.78% of power reduction in Linux kernel 3.10 and 3.6% in Linux kernel 3.14. In the future, the focus of the study should be focusing on further reduction of the power dissipation by exploration of alternative neural network structures such as a Deep Stacking Network [123] and Hybrid Neural Network [146], and on improving learning efficiency with alternative pre-training methods such as Autoencoder aided pre-training and Genetic Algorithm pre-training.

Chapter 6 Conclusions and Recommendations

This thesis proposed an analysis of power dissipation in multi-core/many-core CPUs and furnished novel usage-pattern-based power optimization scheduling schemes. Classical problems and techniques of multi-core processing, task scheduling, power management, heterogeneous architecture, runtime performance monitoring, CPU core temperature power relationship and neural networks are reviewed, and several scheduling techniques are proposed for power consumption optimization in multi-core systems. The proposed methods are successfully applied in the realization of Linux scheduling schemes proposed. The first part of this thesis examines the different contributing components named as the Thread Interference Power (TIP) and Thread Solitary Power (TSP) of the power dissipation in processors and proposed novel approach to mitigate the performance impact of targeting component named as the apps usage pattern based scheduling. The second part of the thesis studies the system performance metric behavior corresponding to the power dissipation patterns by quantifying the Performance Metric Parameters (PMPs) and the Power Dissipation Problems (PDPs). A Linux Scheduler named as the Re Scheduler is designed to mitigate the power dissipation components in system with heterogeneous multi-core architecture. The last part of the thesis examines various applications of different neural network algorithms and applies selected function blocks in a stacked manner to build the Re Deep Function Network (DFN) to reduce the overhead generated by the Re Scheduler and improve its prediction accuracy.

In summary, the thesis makes following contributions:

- Established a power dissipation model of Thread Interference Power and Thread Solitary Power (TIP-TSP) that can be applied to analyze power dissipation pattern in multi-core/many-core system.
- Designed TIP-TSP mitigation approach such as the HOLD, SUPER and DEDICATED scheduling scheme that can be adopted to optimize application scheduling problems.
- Proposed a quantization method for the TIP-TSP model with Performance Monitoring Unit readings and temperature aware CPU energy budget computation approach.
- Adopted the quantized TIP-TSP model based energy budget computation approach in Linux Scheduler and achieved power reduction at least 10.2% and up to 25.4% (Section 4.5) comparing to the native schedulers Linux Global Task Scheduling (GTS) and In Kernel Switcher (IKS) in heterogeneous multi-core system.
- Proposed the structure of a multifunction block constructed deep Neural Network structure and used it to solve time series prediction problems in the proposed Scheduler.
- Examine a series of neural network optimizing techniques and reduced the computation overhead of the network by 50.23% to 84.93% (Section 5.5).

The main contributions of the thesis are summarized and the recommendations for further works are presented in the following sections.

6.1 TIP-TSP Model

The TIP-TSP based power dissipation analytical model proposed in Chapter 3 focuses on the different levels of additional power generated in the execution of multiple application threads. It is aimed to resolve the issues of existing power management schemes of being performance restricted by the triggering parameters chosen. The key restriction to the existing power management is that only a handful of parameters upon making scheduling decisions, such as the average time of the targeted thread being scheduled, the expected operating voltage and frequency growth incurred by scheduling a particular thread are considered. Due to the restriction, the solving of scheduling problems is focused on maximizing the direct cost reduction originated from manipulating the targeted application threads. However in a multithreading environment the amount of power dissipation related to the direct execution of an instruction is very much bounded by the multiplexing operation to parallel application threads. Thus the TIP-TSP model is proposed to solve power dissipation problems with the motive to minimize the indirect cost by the multithreading mechanism.

The proposed TIP-TSP model is adopted in the realization of three app-usage-driven scheduling schemes. (1) HOLD scheme: it keeps threads from selected apps to wait in run queue and executes the app threads in a reordered sequence to reduce power dissipation resulted from *AppSets* with excessive TIP components; (2) DEDICATED scheme: it reserves a run queue of one CPU core to process threads from one app to avoid power dissipating through TIP components; (3) TOTAL PARALLELISM scheme: it arranges the run queues of all CPU cores to share the same *AppSet* scheduling sequences to maximize the throughput of apps developed with multithreading in mind. The TOTAL PARALLELISM scheme mechanism is improved in the Re Scheduler proposed in Chapter 4 and renamed as the SUPER scheme.

The simulation results show a 19% of overall power reduction comparing to the native scheduling schemes (Table 3.6). The results also show that the proposed user app usage pattern based scheduling schemes achieve better power dissipation reduction when there are more TIP components presented in the *AppSet*. Although the performance of the proposed scheduling schemes is better than the state of the art power management techniques, the analysis shows that the amount of Thread Interference Power deducted from the total TIP identified is marginal. Hence to increase the amount of TIP deducted, Dynamic Voltage and Frequency Scaling (DVFS) approach is proposed to comprehend the overhead generated from in computing the power dissipation model at runtime.

6.2 PMP Quantization and PDP Determination

Monitoring accurate power dissipation of a CPU at runtime is a difficult task to accomplish without external power measurement modules installed. Hence in order to accurately address the TIP-TSP power components at runtime, the power dissipated by an application thread in execution is decomposed into features representable by CPU performance events in terms of Performance Metric Parameters (PMPs). Each PMP represents a system event such as cache hit, TLB miss, branch misprediction and context switch. Each PMP is studied and formulated as equations that compute their cost in CPU cycles. The cost of a PMP is generalized with methods

proposed in the past literatures and experiments conducted upon system initialization. Comparing the cost of a hit time PMP such as L1 cache, a miss time PMP such as L1 cache hit normally incurs more cost up to tens or hundreds of its hit time counterpart. The cost of a miss time PMP is architecture dependent. Hence initialization tests are conducted to derive accurate PMP cost. For example, to determine the L2 cache miss PMP cost, numbers of CPU cycles of L1 cache, L2 cache and DRAM access time are gauged using the initialization test.

To quantify the amount of power dissipated by the system, a series of PMP is recorded at the end of every sampling interval. The PMP recorded are then converted to numbers of CPU cycles using the PMP quantization equations derived. Representing the power dissipation TIP-TSP components in number of CPU cycles helps to avoid redundancy in utilizing power figure directly.

In order to have better TIP-TSP prediction error tolerance, DVFS is incorporated to the scheduler schemes proposed in Chapter 3. CPU core voltage and frequency is adjustable to the scheduler to meet the computation requirements of the application threads. The current DVFS techniques are either thermal status aware or power budget aware, which allows the scheduler to make decisions on adjusting the operating frequency and voltage of a CPU core based on the expected power and thermal level of the incoming application thread. The original DVFS technique only manages the frequency and operating pair adjustment and application threads are evenly distributed into different CPU cores for maximum parallelism. However the existing scheduler governed DVFS scheme is inefficient, because the CPU power varies according to the temperature of the core and being completely balance in workloads of all cores does not result in the best energy conservation. Thus the proposed scheduler adopts the energy budget as an improved DVFS decision constraint that considers the influence of both power and thermal dissipation on the performance.

To realize the energy budget based DVFS technique, the scheduler is expected to operate in a way that the performance boost it provided to the system with increased operating frequency and voltage does not exceeds the amount of power dissipation reduction it is able to achieve. In order to convert the amount of power dissipation reduced in Watt directly into the amount of performance boost in frequency and number of intervals, a thermal static and dynamic power model is developed. This model converts operating frequency, voltage and temperature into energy in CPU cycles. The power dissipation reduced by the Re Scheduler converted to CPU cycles using the PMP quantization equations is converted into energy by considering the instantaneous frequency, voltage, temperature, and sampling period using the power dissipation quantization model. The energy budget denotes the accumulated amount of energy saved by the Re Scheduler. The DVFS is aware of the total energy budget available for the system to boost frequency and voltage of the CPU cores, and the proposed scheduler decides the amount of performance boost by computing the energy required to increase the frequency for a sampling period. As a result the proposed scheduling schemes gain more freedom and energy awareness to allocate incoming application threads as comparing the existing native scheduler with the existing DVFS techniques.

There are three scheduling schemes used in Re Scheduler, HOLD, DEDICATED and SUPER. HOLD and DEDICATED are similar to those introduced in Chapter 3, while SUPER is a scheme that increases the time slice length of certain *AppSet* and reduces the time slice length for another to avoid TIP dissipation. The Re

Scheduler determines which scheduling scheme to apply by examining the TIP-TSP model of *AppSets* in the run queue. It is a complicated problem for the scheduler to determine which scheme to apply, as there are 25 PMP values to consider for each *AppSet* in the run queue. Hence a group of Power Dissipation Problems (PDPs) are defined based on power dissipation problems discussed in the existing literatures and the generalization of the test results. There is a total of 9 PDPs under three categories, the cache miss dominated PDPs, context switch dominated PDPs and serial coding bottleneck dominated PDPs. Each PDP is mapped with a given set of scheduling schemes to mitigate the TIP components of the *AppSet*. PDPs can be identified by comparing the PMP value recorded with the threshold value of PDPs. The implementation of PDPs helps to reduce the complexity of determining which scheduling scheme to pick with given PMPs.

The result shows that the proposed scheduler is able to improve the power dissipation reduction up to 22% when comparing with the native Linux schedulers (Table 4.7). However, it is also noticeable that the proposed scheduler also contributes 10% of overall TIP dissipated due to overhead generated in computation and prediction error (Figure 5.1). This is mainly due to the problem dimensionality reduction measure, which is the statistical power dissipation problem to scheduling scheme mapping. To further reduce the overhead and improve the efficiency of the proposed scheduler, non-parametric approximation approaches are explored to comprehend prediction and scheduling scheme mapping mechanism.

6.3 Neural Network Time Series Data Classification and Optimization

Multiple Neural Network algorithms are studied to comprehend the proposed scheduler with better prediction and non-parametric problem solving capacity. One drawback about neural network is that deep structured multilayer network used in solving complex problems has the tendency of being computation intensive at training stage and becomes impractical to realize in mobile computing platform. Hence a dynamic programming approach that combines the table based decision making function of the proposed scheduler and the non-parametric problem solving element of a Neural Network is constructed, where different neural networks are used as the basic function blocks to be used in different computation stages of the scheduling problem. The proposed Re Deep Function Network (DFN) in Chapter 5 consists of 6 layers in total and each layer maps a part of the function of the Re Scheduler proposed in Chapter 4. Such loose coupled block based deep neural network has demonstrated its ability of making multi-staged complex decisions with linear training time growth.

To better the prediction mechanism of the proposed scheduler, the incoming performance events are decoupled and viewed as time series data with different features. The recurrent neural network function blocks are deployed to predict the incoming application thread based on the feature time series. The non-parametric problem to scheduling scheme mapping is resolved with supervised multilayer perceptron.

Optimization technique such as sensitivity analysis, weight pruning and matrix compression have also been utilized to reduce the computation overhead of the network. The neural network aided Re DFN shows an

average computation overhead reduction up to 72.04% (Figure 5.34) and power reduction improvement of 4.69% (Table 5.10) comparing to the Re Scheduler proposed in Chapter 4.

6.4 Future Works

The Re Scheduler is reconstructed into the ANN based Re DFN. The reconstruction brings in an improvement in the computation efficiency as shown in the results presented in Chapter 5. Re DFN's capability of approaching complex general approximation scheduling problems has improved the decision making efficiency and predicting accuracy as discussed in the sections below Figures 5.34 and 5.35. Thus, developing a neural network based scheduler structure is in benefit with the development of power optimization schemes proposed in this work. Should the Scheduler be deployed in a mobile AI chip in the future, the computational overhead of the Re Scheduler would be completely eliminated. To further improve the performance of the proposed scheduler, genetic algorithm initialization approach, different settings of activation functions and Tensorization based data mapping are considered to be implemented in the future. With the aid of the proposed algorithms and technique, the future iterations of the scheduler are likely to be more concise, computation efficient, and achieves higher degree of automation in solving scheduling problems at runtime.

Genetic Algorithm Initialization: genetic algorithm (GA) [147, 148] is a method used to solve optimization problems by modifying a population of proposed solutions. The optimization process of GA reassembles genetic based biological evolution, where the solution of the targeting problem gets evolved according to the output of system at each iteration. There are two main operators used in GA: cross over operator, by performance cross over a new offspring solution gets created by combining different part from existing parent solutions; mutation operator, by performing mutation an existing solution gets improved by partially altering its content. As shown in Figure 6.1, GA can be used to find the optimized solution of a neural network: cross over reproduce a new weight matrices for the given neural network by combing neuron weights from two parent weight matrices; mutation changes the weight matrix of an existing solution, which indicates in the figure as a changing of connector colors from red to green and blue to purple.

Figure 6.1 Genetic algorithm cross over and mutation

The learning approach used in Re DFN is backpropagation, which is believed to be able to outperform GA in problems with a solution space with less local minimal and underperforms it in other cases. As the scheduling

problem model in the Re DFN is likely to possess a complex solution space, hence GA can be considered as the learning algorithm and may help to simplify the structure of Re DFN by removing the Power Dissipation Problem (PDP) layer.

Tensorization data mapping: tensors are multidimensional arrays representing lower dimensioned vectors and matrices with higher dimensioned tensors. Vectors and matrices can be converted to tensors through tensorization techniques [149], where after tensorization the product of a 3-dimensional tensor and a 1-dimensional vector is able to generate a 2-dimensional matrix. Hence it is able to stimulate the performance of neural network layers with tensor elements that processes the same input vector in multiple neural network blocks simultaneously. The Re DFN is constructed with the idea of exploiting the problem solving capability of a deep neural network (DNN) and yet maintains the computation complexity of relative simple neural network. The advantages are prominent as the vanishing gradient problem relating to applying backpropagation algorithm in a DNN does not affect the Re DFN as much. However the degradation of learning efficiency and accuracy in a loosely coupled stacked multilayer neural network can become a big problem if the size of the system is large, which is one of the reasons why Re DFN without undergoing optimization generated more computation overhead compared to Re Scheduler. With the implementation of tensorized layers in the Re DFN, the separated layers are expected to respond to vectors at the input simultaneously instead of processing it in a pipelined manner. As a result, the training complexity of the Re DFN system is expected to be reduced and the system is able to handle tasks in a greater scale without inflicting much computation overheads.

Activation function setting: activation functions resides in the neuron or node of neural networks, and an output is produced if they are triggered. Hence the firing condition of a neuron is not just bounded by the input level and the weightage connecting to it, but also the setup of its activation functions. There is a wide selection of activation functions, and Figure 6.2 addresses three of them: sigmoid, one of the most common activation function used in classification problems; rectified linear unit (ReLU), another popular activation function that performs well in convolutional neural networks (CNNs); hyperbolic tangent, an activation function that performs better comparing to the sigmoid in DNNs. Its negative value response enables it to counter the vanishing gradient problem in DNNs better.

Figure 6.2 Response of different active functions

The choice of activation function can impact the performance of the neural network extensively. The effect not only reflects on their affinity with different neural network setup but can also vary according to the format of

training data, the size of neural network, number of neurons and number of layers. Hence the determination process of activation function for a neural network can be regarded as a trial and error process. An activation function comparison experiment can be conducted for Re DFN to further increase its computation efficiency and reduce the overhead generated in the learning process.

Author's Publications

Hou Zhao Qi, Jong Ching Chuen, and Andreas Herkersdorf. "Linux apps-usage-driven power dissipation-aware scheduler." 2016 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2016.

Hou Zhao Qi, Jong Ching Chuen, and Andreas Herkersdorf. "ANN Predicted Apps-Usage Aware Linux Scheduler for Asymmetrical Multi Cluster SoC." 2016 5th International Conference on Software and Information Engineering (ICSIE 2016)

Hou Zhao Qi, Jong Ching Chuen, and Andreas Herkersdorf. "ANN Predicted Apps-Usage Aware Linux Scheduler for Asymmetrical Multi Cluster SoC." Journal of Software 11.7 (2016): 623-630.

Hou Zhao Qi, Jong Ching Chuen, and Andreas Herkersdorf. "Apps-usage driven energy management for multi-core mobile computing systems." 2014 International Symposium on Integrated Circuits (ISIC). IEEE, 2014.

Reference

- [1] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch, "Power management of online data-intensive services," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, 2011, pp. 319-330.
- [2] William Lloyd Bircher and Lizy John, "Predictive power management for multi-core processors," in *International Symposium on Computer Architecture*, 2010, pp. 243-255.
- [3] Andrea Bartolini, Matteo Cacciari, Andrea Tilli, and Luca Benini, "Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 170-183, 2013.
- [4] Liang Xia, Yongxin Zhu, Jun Yang, Jingwei Ye, and Zonghua Gu, "Implementing a thermal-aware scheduler in linux kernel on a multi-core processor," *The Computer Journal*, vol. 53, pp. 895-903, 2010.
- [5] David Tam, Reza Azimi, and Michael Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *ACM SIGOPS Operating Systems Review*, 2007, pp. 47-58.
- [6] Hou Zhao Qi Rex, Jong Ching Chuen, and Andreas Herkersdorf, "Apps-usage driven energy management for multicore mobile computing systems," in *2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 472-475.
- [7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley 2006.
- [8] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer, "High performing cache hierarchies for server workloads: relaxing inclusion to capture the latency benefits of exclusive caches," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 343-353.
- [9] Jason M Allred, Sanghamitra Roy, and Koushik Chakraborty, "Dark silicon aware multicore systems: Employing design automation with architectural insight," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 1192-1196, 2014.
- [10] Shih-Hao Ou, Che-Wei Yeh, Tai-Jyi Lin, and Chih-Wei Liu, "A smart stream controller for efficient implementation of streaming applications on the heterogeneous multicore processor," in *2012 IEEE International Symposium on Circuits and Systems*, 2012, pp. 1335-1338.
- [11] Li Luo, Hongjun He, Qiang Dou, and Weixia Xu, "Hardware/Software partitioning for heterogeneous multicore SoC using genetic algorithm," in *Intelligent System Design and Engineering Application (ISDEA), 2012 Second International Conference on*, 2012, pp. 1267-1270.
- [12] Bharathwaj Raghunathan, Yatish Turakhia, Siddharth Garg, and Diana Marculescu, "Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 39-44.
- [13] N. Almoosa, W. Song, Y. Wardi, and S. Yalamanchili, "A power capping controller for multicore processors," in *American Control Conference (ACC), 2012*, 2012, pp. 4709-4714.
- [14] Koushik Chakraborty and Sanghamitra Roy, "Architecturally homogeneous power-performance heterogeneous multicore systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 670-679, 2013.
- [15] Tejaswini Kolpe, Antonia Zhai, and Sachin S Sapatnekar, "Enabling improved power management in multicore processors through clustered DVFS," in *2011 Design, Automation & Test in Europe*, 2011, pp. 1-6.
- [16] Paul Bassett and Martin Saint-Laurent, "Energy efficient design techniques for a digital signal processor," in *2012 IEEE International Conference on IC Design & Technology*, 2012, pp. 1-4.
- [17] Wei Chen, Fengqian Gao, and Yuan Lu, "Policy based power management in cloud environment with intel intelligent power node manager," in *2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops*, 2012, pp. 66-69.
- [18] Zheng Zhou, Junjun Gu, and Gang Qu, "Scheduling for multi-core processor under process and temperature variation," in *Embedded Multicore Socs (MCSoc), 2012 IEEE 6th International Symposium on*, 2012, pp. 113-120.
- [19] Buyoung Yun, Kang G Shin, and Shige Wang, "Thermal-aware scheduling of critical applications using job migration and power-gating on multi-core chips," in *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011, pp. 1083-1090.

- [20] Jun Lu and Qinru Qiu, "Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting," in *Green Computing Conference and Workshops (IGCC), 2011 International*, 2011, pp. 1-6.
- [21] Niti Madan, Alper Buyuktosunoglu, Pradip Bose, and Murali Annavaram, "A case for guarded power gating for multi-core processors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 291-300.
- [22] Pritesh Vora and Masud H Choudhary, "Prospects and implementation of Non-DVFS dynamic thermal management techniques," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 53-56.
- [23] Inseok Choi, Minshu Zhao, Xu Yang, and Donald Yeung, "Experience with improving distributed shared cache performance on tilera's tile processor," *IEEE Computer Architecture Letters*, vol. 10, pp. 45-48, 2011.
- [24] Enrico Bini, Giorgio Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, *et al.*, "Resource management on multicore systems: The ACTORS approach," *IEEE Micro*, vol. 31, pp. 72-81, 2011.
- [25] Christophe Tretz, Chen Guo, and Lawrence Jacobowitz, "An all digital frequency-locked loop immune to hysteresis effects for power management of multicore processors," in *SOI Conference (SOI), 2010 IEEE International*, 2010, pp. 1-2.
- [26] W Lloyd Bircher and Lizy Kurian John, "Core-level activity prediction for multicore power management," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, pp. 218-227, 2011.
- [27] Anca Molnos and Kees Goossens, "Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors," in *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, 2009, pp. 409-418.
- [28] Joonho Kong, Jinhang Choi, Lynn Choi, and Sung Woo Chung, "Low-cost application-aware DVFS for multi-core architecture," in *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, 2008, pp. 106-111.
- [29] Lichen Weng, Gang Quan, and Chen Liu, "PCOUNT: a power aware fetch policy in simultaneous multithreading processors," in *Green Computing Conference and Workshops (IGCC), 2011 International*, 2011, pp. 1-6.
- [30] Arvind Kandhalu, Junsung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar, "Energy-aware partitioned fixed-priority scheduling for chip multi-processors," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011, pp. 93-102.
- [31] Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel, "Accurate mutlicore processor power models for power-aware resource management," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 419-426.
- [32] Hwisung Jung and Massoud Pedram, "Improving the efficiency of power management techniques by using Bayesian classification," in *9th International Symposium on Quality Electronic Design (isqed 2008)*, 2008, pp. 178-183.
- [33] Hwisung Jung and Massoud Pedram, "Supervised learning based power management for multicore processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 1395-1408, 2010.
- [34] Masnida Hussin, Young Choon Lee, and Albert Y Zomaya, "Priority-based scheduling for large-scale distribute systems with energy awareness," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 503-509.
- [35] Gaurav Dhiman, "System-level power management using online learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 676-689, 2009.
- [36] Priyankar Ghosh, Aritra Hazra, Rahul Gonnabhaktula, Niraj Bhilegaonkar, Pallab Dasgupta, Chittaranjan Mandal, *et al.*, "POWER-SIM: An SOC Simulator for Estimating Power Profiles of Mobile Workloads," *Journal of Low Power Electronics*, vol. 8, pp. 293-303, 2012.
- [37] Joon-Myung Kang, Sin-seok Seo, and James Won-Ki Hong, "Usage pattern analysis of smartphones," in *Network Operations and Management Symposium (APNOMS), 2011 13th Asia-Pacific*, 2011, pp. 1-8.
- [38] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors," in *Green Computing Conference, 2010 International*, 2010, pp. 123-133.
- [39] Milind B Kamble and Kanad Ghose, "Analytical energy dissipation models for low power caches," in *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, 1997, pp. 143-148.

- [40] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula, "Fusion: design tradeoffs in coherent cache hierarchies for accelerators," in *ACM SIGARCH Computer Architecture News*, 2015, pp. 733-745.
- [41] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 25-37.
- [42] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 343-355.
- [43] T Puzak, Allan Hartstein, P Emma, and Viji Srinivasan, "Measuring the cost of a cache miss," in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2006.
- [44] Leo C Singleton, Christian Poellabauer, and Karsten Schwan, "Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling," in *Electronic Imaging 2005*, 2005, pp. 121-125.
- [45] Stijn Eyerman, James E Smith, and Lieven Eeckhout, "Characterizing the branch misprediction penalty," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 48-58.
- [46] Amit Gandhi, Haitham Akkary, and Srikanth T Srinivasan, "Reducing branch misprediction penalty via selective branch recovery," in *Software, IEE Proceedings-*, 2004, pp. 254-264.
- [47] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 623-634.
- [48] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift, "BadgerTrap: a tool to instrument x86-64 TLB misses," *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 20-23, 2014.
- [49] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, *et al.*, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," in *ACM SIGARCH Computer Architecture News*, 2015, pp. 79-91.
- [50] Chuanpeng Li, Chen Ding, and Kai Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, p. 2.
- [51] Francis M David, Jeffrey C Carlyle, and Roy H Campbell, "Context switch overheads for Linux on ARM platforms," in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, p. 3.
- [52] Fang Liu, Fei Guo, Yan Solihin, Seongbeom Kim, and Abdulaziz Eker, "Characterizing and modeling the behavior of context switch misses," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 91-101.
- [53] Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev, "The Effect of NUMA Tunings on CPU Performance," in *Journal of Physics: Conference Series*, 2015, p. 092010.
- [54] Zhenyu Qi, Brett H Meyer, Wei Huang, Robert J Ribando, Kevin Skadron, and Mircea R Stan, "Temperature-to-power mapping," in *Computer Design (ICCD), 2010 IEEE International Conference on*, 2010, pp. 384-389.
- [55] Ramkumar Jayaseelan and Tulika Mitra, "Temperature aware task sequencing and voltage scaling," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 618-623.
- [56] Kevin Skadron, Mircea R Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan, "Temperature-aware microarchitecture," in *ACM SIGARCH Computer Architecture News*, 2003, pp. 2-13.
- [57] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan, "HotSpot: A compact thermal modeling methodology for early-stage VLSI design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 501-513, 2006.
- [58] R Arvind and Mrs A Vinnarasi, "Thermal Data Acquisition on a Quad Core Machine."
- [59] Ehsan K Ardestani, Amirkoushyar Ziabari, Ali Shakouri, and Jose Renau, "Enabling power density and thermal-aware floorplanning," in *Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM), 2012 28th Annual IEEE*, 2012, pp. 302-307.
- [60] B. Salami, M. Baharani, H. Noori, and F. Mehdipour, "Physical-aware task migration algorithm for dynamic thermal management of SMT multi-core processors," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 292-297.
- [61] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, *et al.*, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of*

- the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012, pp. 317-328.
- [62] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski Jr, Thomas F Wenisch, and Scott Mahlke, "Heterogeneous microarchitectures trump voltage scaling for low-power cores," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 237-250.
 - [63] Andrew B Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori, "Enhancing the efficiency of energy-constrained DVFS designs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 1769-1782, 2013.
 - [64] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa, "A framework for application-guided task management on heterogeneous embedded systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, p. 42, 2016.
 - [65] Vincent Nollet, Prabhat Avasare, J-Y Mignolet, and Diederik Verkest, "Low cost task migration initiation in a heterogeneous mp-soc," in *Design, Automation and Test in Europe*, 2005, pp. 252-253.
 - [66] ARM Linaro, "big.LITTLE Technology The Future of Mobile," https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf, p. 12, 2013.
 - [67] Peter Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White paper*, pp. 1-8, 2011.
 - [68] Mathieu Poirier, "In Kernel Switcher: A solution to support ARM's new big. LITTLE technology," in *Embedded Linux Conference (ELC'13)*, 2013.
 - [69] ARM Linaro, "Energy Aware Scheduling Linaro," <https://developer.arm.com/open-source/energy-aware-scheduling>, 2015.
 - [70] Atul Negi and P Kishore Kumar, "Applying machine learning techniques to improve linux process scheduling," in *TENCON 2005-2005 IEEE Region 10 Conference*, 2005, pp. 1-6.
 - [71] George Anderson, Tshilidzi Marwala, and Fulufhelo Vincent Nelwamondo, "Multicore scheduling based on learning from optimization models," *Int. J. Innovative Comput. Inform. Control.* v9 i4, pp. 1511-1522, 2013.
 - [72] Chee Peng Lim and Robert F. Harrison, "Online pattern classification with multiple neural network systems: an experimental study," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 33, pp. 235-247, 2003.
 - [73] Yu Zhang, Ekapol Chuangsuwanich, and James R Glass, "Extracting deep neural network bottleneck features using low-rank matrix factorization," in *ICASSP*, 2014, pp. 185-189.
 - [74] Asja Fischer and Christian Igel, "Training restricted Boltzmann machines: An introduction," *Pattern Recognition*, vol. 47, pp. 25-39, 2014.
 - [75] Geoffrey E Hinton, "Learning multiple layers of representation," *Trends in cognitive sciences*, vol. 11, pp. 428-434, 2007.
 - [76] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng, "Multimodal deep learning," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 689-696.
 - [77] B Chandra and Rajesh K Sharma, "Exploring autoencoders for unsupervised feature selection," in *2015 International Joint Conference on Neural Networks (IJCNN)*, 2015, pp. 1-6.
 - [78] Alireza Makhzani and Brendan Frey, "k-Sparse Autoencoders," *arXiv preprint arXiv:1312.5663*, 2013.
 - [79] Vukosi N Marivate, Fulufhelo V Nelwamondo, and Tshilidzi Marwala, "Autoencoder, principal component analysis and support vector regression for data imputation," *arXiv preprint arXiv:0709.2506*, 2007.
 - [80] Telmo Amaral, Luís M Silva, Luís A Alexandre, Chetak Kandaswamy, Jorge M Santos, and Joaquim Marques de Sá, "Using different cost functions to train stacked auto-encoders," in *Artificial Intelligence (MICAI), 2013 12th Mexican International Conference on*, 2013, pp. 114-120.
 - [81] Hugo Larochelle and Iain Murray, "The Neural Autoregressive Distribution Estimator," in *AISTATS*, 2011, p. 2.
 - [82] Li Deng, Brian Hutchinson, and Dong Yu, "Parallel Training for Deep Stacking Networks," in *INTERSPEECH*, 2012, pp. 2598-2601.
 - [83] George G Lendaris, "Adaptive dynamic programming approach to experience-based systems identification and control," *Neural networks*, vol. 22, pp. 822-832, 2009.
 - [84] Pierre Baldi and Kurt Hornik, "Neural networks and principal component analysis: Learning from examples without local minima," *Neural networks*, vol. 2, pp. 53-58, 1989.

- [85] Hung-Che Chen Jyh-Da Wei, "Using neural networks for evaluation in heuristic search algorithm," 2011.
- [86] Min Yang, Wenting Tu, Wenpeng Yin, and Ziyu Lu, "Deep Markov Neural Network for Sequential Data Classification," *Volume 2: Short Papers*, p. 32.
- [87] Philippe Leray and Patrick Gallinari, "Feature selection with neural networks," *Behaviormetrika*, vol. 26, pp. 145-166, 1999.
- [88] Jian-Bo Yang, Kai-Quan Shen, Chong-Jin Ong, and Xiao-Ping Li, "Feature selection for mlp neural network: The use of random permutation of probabilistic outputs," *IEEE Transactions on Neural Networks*, vol. 20, pp. 1911-1922, 2009.
- [89] Georg Dorffner, "Neural networks for time series processing," in *Neural Network World*, 1996.
- [90] Fabian Mörchen, "Time series feature extraction for data mining using DWT and DFT," ed: Univ., 2003.
- [91] James Martens, "Deep learning via Hessian-free optimization," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 735-742.
- [92] James Martens and Ilya Sutskever, "Learning recurrent neural networks with hessian-free optimization," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1033-1040.
- [93] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio, "How to construct deep recurrent neural networks," *arXiv preprint arXiv:1312.6026*, 2013.
- [94] Naoto Shiraga, Seiichi Ozawa, and Shigeo Abe, "A reinforcement learning algorithm for neural networks with incremental learning ability," in *Neural Information Processing, 2002. ICONIP'02. Proceedings of the 9th International Conference on*, 2002, pp. 2566-2570.
- [95] Siva Reddy Gangireddy, Fergus McInnes, and Steve Renals, "Feed forward pre-training for recurrent neural network language models," in *INTERSPEECH*, 2014, pp. 2620-2624.
- [96] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1-7, 2011.
- [97] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469-480.
- [98] Trinh Minh Tri Do, Jan Blom, and Daniel Gatica-Perez, "Smartphone usage in the wild: a large-scale analysis of applications and context," in *Proceedings of the 13th international conference on multimodal interfaces*, 2011, pp. 353-360.
- [99] Zongwei Zhu, Beilei Sun, Xi Li, and Xuehai Zhou, "Application-aware group scheduler for Android," in *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*, 2014, pp. 43-49.
- [100] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley, "OOPSLA 2002: Reconsidering custom memory allocation," *ACM SIGPLAN Notices*, vol. 48, pp. 46-57, 2013.
- [101] Andreas Weissel and Frank Bellosa, "Process cruise control: event-driven clock scaling for dynamic power management," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002, pp. 238-246.
- [102] Yongbing Huang, Zhongbin Zha, Mingyu Chen, and Lixin Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014, pp. 45-54.
- [103] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakash Srivastava, *et al.*, "Stash: Have your scratchpad and cache it too," in *ACM SIGARCH Computer Architecture News*, 2015, pp. 707-719.
- [104] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, 2014, pp. 269-284.
- [105] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture*, 2006, pp. 347-358.
- [106] Myungsun Kim, Kibeom Kim, James R Geraci, and Seongsoo Hong, "Utilization-aware load balancing for the energy efficient operation of the big. LITTLE processor," in *Proceedings of the conference on Design, Automation & Test in Europe*, 2014, p. 223.
- [107] Sigurd Israelsson, "Energy Efficiency Analysis of ARM big. LITTLE Global Task Scheduling," 2015.

- [108] ARM Cortex, "A57 MPCore Processor Technical Reference Manual Infocenter. arm. com Arithmetic," *Logical Unit Advanced SIMD Micro-Operation Vector Floating Point*.
- [109] Intel Intel, "and IA-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, vol. 1, p. 64.
- [110] Arnaldo Carvalho de Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, 2010.
- [111] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade, "Architectural Impact on Performance of In-memory Data Analytics: Apache Spark Case Study," *arXiv preprint arXiv:1604.08484*, 2016.
- [112] Tatsuhiko Chiba and Tamiya Onodera, "Workload Characterization and Optimization of TPC-H Queries on Apache Spark," IBM Research Report, Tokyo, RT09682015.
- [113] Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eli Weissmann, "Power management architecture of the 2nd generation Intel® Core™ microarchitecture, formerly codenamed Sandy Bridge," in *Hot Chips*, 2011, p. 0.
- [114] Qianwei Kuang, Hongxia Liu, Wen Zhou, Bo Gao, Aaron Zhao, and Sai Tallavarjula, "Transport mechanism of the leakage current in MIS capacitor with HfO₂/SiO₂ stack gate," in *Electron Devices and Solid-State Circuits, 2009. EDSSC 2009. IEEE International Conference of*, 2009, pp. 229-232.
- [115] Kevin Skadron, Mircea R Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, pp. 94-125, 2004.
- [116] I Molnar, "Linux cfs scheduler," ed, 2007.
- [117] Hou Zhao Qi Rex, Jong Ching Chuen, and Andreas Herkersdorf, "ANN Predicted Apps-Usage Aware Linux Scheduler for Asymmetrical Multi Cluster SoC," *JSW*, vol. 11, pp. 623-630, 2016.
- [118] Samir Kumar, "Introducing Qualcomm Zeroth Processors: Brain-Inspired Computing," ed: Qualcomm, 2013.
- [119] Iris Hendrickx and Antal Van Den Bosch, "Hybrid algorithms with instance-based classification," in *European Conference on Machine Learning*, 2005, pp. 158-169.
- [120] D Randall Wilson and Tony R Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural Networks*, vol. 16, pp. 1429-1451, 2003.
- [121] Kyunghyun Cho, "Boltzmann machines and denoising autoencoders for image denoising," *arXiv preprint arXiv:1301.3468*, 2013.
- [122] Li Deng, Michael L Seltzer, Dong Yu, Alex Acero, Abdel-rahman Mohamed, and Geoffrey E Hinton, "Binary coding of speech spectrograms using a deep auto-encoder," in *Interspeech*, 2010, pp. 1692-1695.
- [123] Chris Häusler and Alex Susemihl, "Temporal Autoencoding Restricted Boltzmann Machine," *arXiv preprint arXiv:1210.8353*, 2012.
- [124] Brian Hutchinson, Li Deng, and Dong Yu, "Tensor deep stacking networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 1944-1957, 2013.
- [125] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio, "On the difficulty of training recurrent neural networks," *ICML (3)*, vol. 28, pp. 1310-1318, 2013.
- [126] Paul Smolensky, "Information processing in dynamical systems: Foundations of harmony theory," DTIC Document 1986.
- [127] Asja Fischer and Christian Igel, "An introduction to restricted Boltzmann machines," in *Iberoamerican Congress on Pattern Recognition*, 2012, pp. 14-36.
- [128] Geoffrey E Hinton, "Training products of experts by minimizing contrastive divergence," *Neural computation*, vol. 14, pp. 1771-1800, 2002.
- [129] André Stuhlsatz, Jens Lippel, and Thomas Zielke, "Discriminative feature extraction with deep neural networks," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 2010, pp. 1-8.
- [130] Fernando Pérez-Cruz, "Kullback-Leibler divergence estimation of continuous distributions," in *2008 IEEE international symposium on information theory*, 2008, pp. 1666-1670.
- [131] Ilya Sutskever, James Martens, and Geoffrey E Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1017-1024.
- [132] Derong Liu and Zeng-Guang Hou, *Advances in Neural Networks-ISNN 2007: 4th International Symposium on Neural Networks, ISNN 2007 Nanjing, China, June 3-7, 2007. Proceedings* vol. 3: Springer Science & Business Media, 2007.
- [133] Haibin Cheng, Pang-Ning Tan, Jing Gao, and Jerry Scripps, "Multistep-ahead time series prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2006, pp. 765-774.

- [134] Matthias Scholz, Martin Fraunholz, and Joachim Selbig, "Nonlinear principal component analysis: neural network models and applications," in *Principal manifolds for data visualization and dimension reduction*, ed: Springer, 2008, pp. 44-67.
- [135] David G Garson, "Interpreting neural network connection weights," 1991.
- [136] ATC Goh, "Back-propagation neural networks for modeling complex systems," *Artificial Intelligence in Engineering*, vol. 9, pp. 143-151, 1995.
- [137] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, "Learning internal representations by error propagation," DTIC Document 1985.
- [138] Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang W Koh, Quoc V Le, and Andrew Y Ng, "Tiled convolutional neural networks," in *Advances in neural information processing systems*, 2010, pp. 1279-1287.
- [139] Robert Leaman, Ritu Khare, and Zhiyong Lu, "Challenges in clinical natural language processing for automated disorder normalization," *Journal of biomedical informatics*, vol. 57, pp. 28-37, 2015.
- [140] Steven J Nowlan and Geoffrey E Hinton, "Simplifying neural networks by soft weight-sharing," *Neural computation*, vol. 4, pp. 473-493, 1992.
- [141] Moo K Chung, "Sparse Network Modeling," 2012.
- [142] Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas, "ACDC: A structured efficient linear layer," *arXiv preprint arXiv:1511.05946*, 2015.
- [143] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis, "Sparse matrix storage format," in *Proceedings of the 16th annual workshop on circuits, systems and signal processing*, 2005, pp. 445-448.
- [144] Ivan P Stanimirovic and Milan B Tasic, "Performance comparison of storage formats for sparse matrices," *FACTA UNIVERSITATIS (NIS) Ser. Math. Inform*, vol. 24, pp. 39-51, 2009.
- [145] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems, 2015," *Software available from tensorflow.org*, vol. 1, 2015.
- [146] Li Wan, Leo Zhu, and Rob Fergus, "A Hybrid Neural Network-Latent Topic Model."
- [147] Steven A Harp and Tariq Samad, "Optimizing neural networks with genetic algorithms," in *Proceedings of the 54th American Power Conference, Chicago*, 2013.
- [148] Duc Pham and Dervis Karaboga, *Intelligent optimisation techniques: genetic algorithms, tabu search, simulated annealing and neural networks*: Springer Science & Business Media, 2012.
- [149] Alexander Novikov, Dmitrii Podoprikin, Anton Osokin, and Dmitry P Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 442-450.
- [150] Pabla, Chandandeep Singh. "Completely fair scheduler." *Linux Journal* 2009.184 (2009): 4.
- [151] Microsoft, "Bot Framework", <https://dev.botframework.com/>, 2016