UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Electronics and Computer Science

Runtime Energy Management of Concurrent Applications for Multi-core Platforms

by

Karunakar Reddy Basireddy

Thesis submitted for the degree of Doctor of Philosophy

April 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Electronics and Computer Science

Doctor of Philosophy

RUNTIME ENERGY MANAGEMENT OF CONCURRENT APPLICATIONS FOR MULTI-CORE PLATFORMS

by Karunakar Reddy Basireddy

Multi-core platforms are employing a greater number of heterogeneous cores and resource configurations to achieve energy-efficiency and high performance. These platforms often execute applications with different performance constraints concurrently, which contend for resources simultaneously, thereby generating varying workload and resources demands over time. There is a little reported work on runtime energy management of concurrent execution, focusing mostly on homogeneous multi-cores and limited application scenarios. This thesis considers both homogeneous and heterogeneous multi-cores and broadens application scenarios. The following contributions are made in this thesis.

Firstly, this thesis presents online Dynamic Voltage and Frequency Scaling (DVFS) techniques for concurrent execution of single-threaded and multi-threaded applications on homogeneous multi-cores. This includes an experimental analysis and deriving metrics for efficient online workload classification. The DVFS level is proactively set through predicted workload, measured through Memory Reads Per Instruction. The analysis also considers thread synchronisation overheads, and underlying memory and DVFS architectures. Average energy savings of up to 60% are observed when evaluated on three different hardware platforms (Odroid-XU3, Intel Xeon E5-2630, and Xeon Phi 7620P).

Next, an energy efficient static mapping and DVFS approach is proposed for heterogeneous multi-core CPUs. This approach simultaneously exploits different types of cores for each application in a concurrent execution scenario. It first selects performance meeting mapping (no. of cores and type) for each application having minimum energy consumption using offline results. Then online DVFS is applied to adapt to workload and performance variations. Compared to recent techniques, the proposed approach has an average of 33% lower energy consumption when validated on the Odroid-XU3.

To eliminate dependency on the offline application profiling and to adapt to dynamic application arrival/completion, an adaptive mapping approach coupled with DVFS is presented. This is achieved through an accurate performance model, and an energy efficient resource selection technique and a resource manager. Experimental evaluation on the Odroid-XU3 shows an improvement of up to 28% in energy efficiency and 7.9% better prediction accuracy by performance models.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

ACPI	Advanced Configuration and Power Interface
AsAP	Asynchronous Array of Simple Processors
BIOS	Basic Input/Output System
CFA	Continuous Frequency Adjustment
CFD	Computational Fluid Dynamics
CP	Combination Point
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DLP	Data-Level Parallelism
DP	Design Point
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
DSP	Digital Signal Processors
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
DyPO	Dynamic Pareto-Optimal
ES	Exhaustive Search
EWMA	Exponential Weighted Moving Average
FPGA	Field-Programmable Gate Array
GPP	General Purpose Processors
GPU	Graphics Processing Unit
HMP	Heterogeneous Multi-Processing
HIBI	Heterogeneous IP Block Interconnection
ILP	Instruction-Level Parallelism
I/O	Input/Output
IP	Intellectual Property
IPC	Instructions Per Cycle
IPS	Instructions Per Second
ISA	Instruction-Set Architecture
ITMD	Inter-cluster Thread-to-core Mapping and DVFS
ITRS	International Technology Roadmap for Semiconductors
IVP	Initial Value Problem
LLC	Last Level Cache

MAE	Mean Absolute Error
MAPC	Memory Accesses Per Cycle
MAPE	Mean Absolute Percentage Error
MAPI	Memory Accesses Per Instruction
MAPM	Memory Accesses Per Micro-operation
MDE	Multicore Development Environment
MIC	Intel Many Integrated Core
MIPS	Microprocessor without Interlocked Pipelined Stages
MLP	Memory-Level Parallelism
MLR	Multinomial Linear Regression
MPSoC	Multi-Processor System-on-Chip
MPSS	Manycore Platform Software Stack
MSR	Model Specific Register
MRPC	Memory Reads Per Cycle
MRPI	Memory Reads Per Instruction
NP	Non-deterministic Polynomial time
NUMA	Non-Uniform Memory Access
OoO	Out-of-Order
PC	Personal Computers
PCA	Principle Component Analysis
PCIe	Peripheral Component Interconnect express
PDA	Personal Digital Assistant
PE	Processing Element
PID	Proportional-Integral-Derivative
PIE	Performance Impact Estimation
PMC	Performance Monitoring Counter
PMU	Performance Monitoring Unit
PPW	Performance Per Watt
QoS	Quality of Service
QPI	Quick Path Interconnect
RAW	Raw Architecture Workstation
RISC	Reduced Instruction Set Computer
RL	Reinforcement Learning
RMS	Recognition, Mining, and Synthesis
ROI	Region Of Interest
RTEM	Runtime Energy Management
RTL	Register Transfer Level
RTM	Runtime Manager
SMT	Simultaneous Multi-threading
SoC	System-on-Chip
SPEC	Standard Performance Evaluation Corporation

$TRIPS$	Tera-op, Reliable, Intelligently adaptive Processing System
VFS	Voltage and Frequency Scaling
WED	Weighted Euclidean Distance
$WEKA$	Waikato Environment for Knowledge Analysis
WMI	Workload Memory-Intensity

Nomenclature

a_i	Actual workload
$Apps_{prfr}$	Performance requirements of applications
CA_{Apps}	Concurrently active applications
$C_{i\_cores_m}$	Used C_i type cores for an application m
C_L	Load capacitance [F]
D_{A_m}	Design points of an application
δ, ζ	Effect of memory contention on performance (latency)
E_i	Cluster with cores of type C_i
$Energy_m$	Energy consumption of an application m [J]
E_o	Energy overhead [J]
E_T	Total energy consumption [J]
η	Speedup
f	Operating frequency [Hz]
f_{max}	Maximum operating frequency [Hz]
γ	Smoothing factor
λ	Performance loss
l_i	Number of cores of type C_i
μ-ops	Micro-operations retired
M_i	MAPM of processing core
m_l	Accesses to local memory
m_r	Accesses to remote memory
$NrCApps$	Number of active applications
n_L	Number of LITTLE cores
n_b	Number of big cores
$P_{dynamic}$	Dynamic power consumption [W]
p_i	Predicted workload
PMC_o	PMC data collection overhead [s]
P_{static}	Static power consumption [W]

P_T	Total power consumption [W]
r	Number of times adaptation is paused
R	Number of threads of an application
TC_{map}	Number of thread-to-core mappings
TC_{map_VF}	Total number of resource combinations
t_e	Execution time [s]
θ	Measure of latency associated with the remote memory access
T_{map}	Thread-to-core-mapping overhead [s]
T_o	Total overhead [s]
T_{sample}	Time interval [s]
$T_{V\text{-}f}$	Overhead associated with *V-f* pair selection [s]
V	Supply voltage [V]
VfS_o	*V-f* switching overhead [s]
W	Workload
W_{E_i}	Workload on a cluster E_i

Declaration of Authorship

I, Karunakar Reddy Basireddy , declare that the thesis entitled *Runtime Energy Management of Concurrent Applications for Multi-core Platforms* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University;
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- Where I have consulted the published work of others, this is always clearly attributed;
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- Parts of this work that have been published are listed under the Research Contributions.

Signed:.....

Date:.....

Acknowledgements

I would like to take this opportunity to extend my thanks to many people who directly and indirectly supported me during my PhD. First of all, my heartfelt gratitude goes to my supervisors; Dr. Geoff V. Merrett for his invaluable guidance, support, and suggestions throughout my PhD and Prof. Bashir M. Al-Hashimi for his direction and advice at many times, and for providing me with an opportunity to undertake this research. In spite of being busy with many responsibilities, they have always found time for regular discussions with me. I would also like to express my thanks to Dr. Amit Kumar Singh for his insightful and stimulating discussions and suggestions. Without their continuous support, this thesis would not have been in its present form.

Many thanks also go to the Engineering and Physical Sciences Research Council (EPSRC) funded PRiME Project for financially supporting me throughout the PhD. I also extend my thanks to the School of Electronics and Computer Science, the University of Southampton for providing state-of-the-art research facilities. Being part of the PRiME project, I had the opportunity to work with a unique group of colleagues: Dr. Dwaipayan Biswas, Dr. Domenico Balsamo, Dr. Eduardo Wachter, Matt Walker, Charles Leech, and Dr. Graeme Bragg to name a few. Thanks to all of them for creating a vibrant research environment for thought-provoking discussions and knowledge sharing. Thanks to Kath Kerr for her kind administrative and logistics support. I would also like to thank Dr. Amit Acharyya for always providing me with career guidance and support.

Further, my sincere thanks to all my friends who have always been there to help and for making my stay in Southampton a pleasant memory. In this regard, I would like to mention a few notable names who are part of our regular discussions at tea breaks/gym sessions: Charan, Satya, Rakshith, Ashwini, Amin, Alberto, Oktay, Cai, Sadegh and many more.

Finally, greatest thanks go to my family (Vengamma, Srinivasulu Reddy, Archana and Duggaiah) and Shri Krishna for their infinite love, believing in me and making me into what I am today.

Chapter 1

Introduction

Computing systems have emerged in many fields (e.g., medical and business) and the demand for higher performance is ever increasing. Performance is usually defined as the execution time of an application, instructions per second, or instructions per cycle [10, 19, 20]. In the '80s and '90s, frequency scaling helped in meeting the increasing demands for performance. Shortening the critical path and exploiting instruction level parallelism allowed the CPU to run at higher clock speeds to improve throughput [21]. Consequently, processor manufacturers were able to double single-threaded performance approximately every 18 months [22], which has added an increased amount of complex logic to the processor core. For a long time, new process technologies allowed for smaller and less energy consuming transistors. Furthermore, techniques such as pipelining, superscalarity and Out-of-Order (OoO) execution were proposed to improve performance

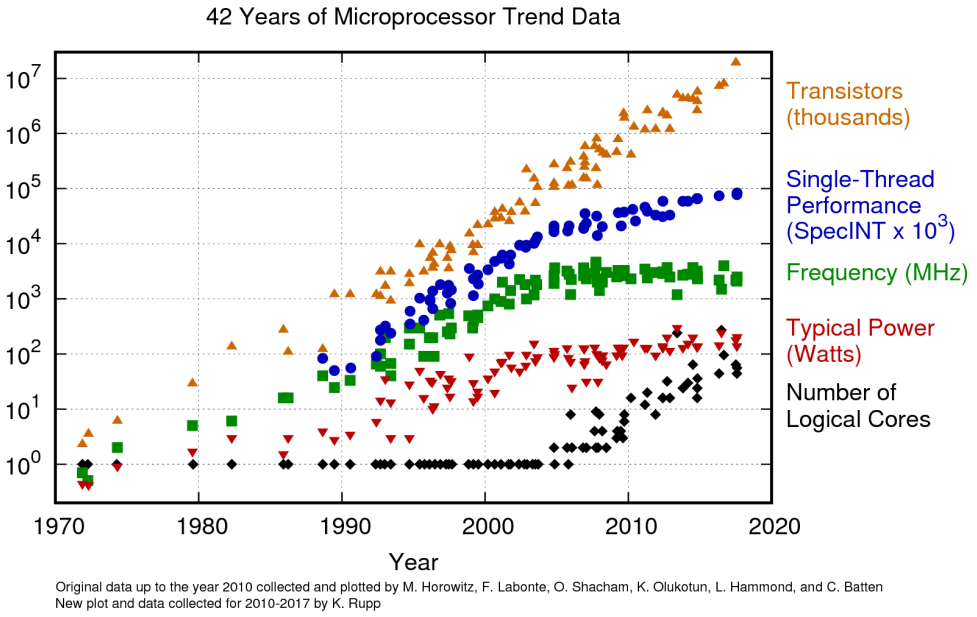

FIGURE 1.1: Historical trends in CPU performance, reprinted from [1]

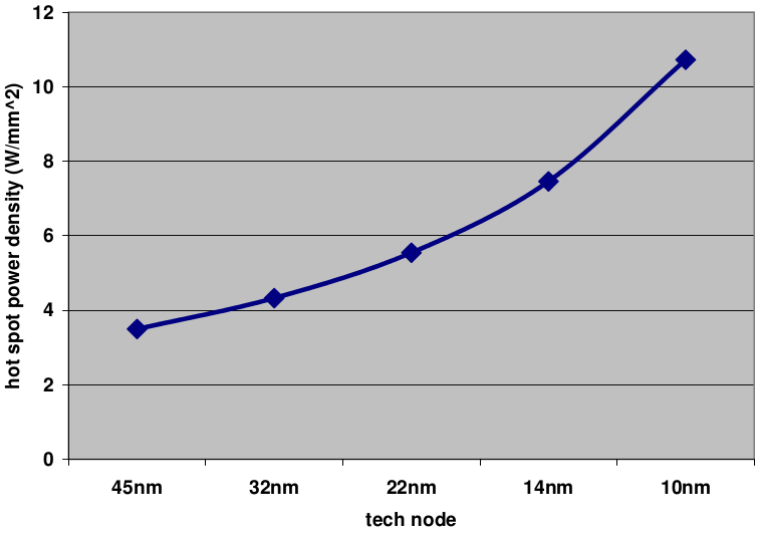

FIGURE 1.2: Hot spot power density scaling with feature size. Reprinted from [2]

[23]. However, as Dennard scaling [24,25] came to an end, increasing gate count on a die and scaling the frequency up to meet the performance demands is no longer a favorable way as heat generation became a serious problem.

As a result, the rapid increase in single-threaded performance during the last few decades seems to have come to an end. Fig. 1.1 shows how Moore's law has been continuing over time [21, 22]. As can be seen from the figure, the transistor count on a die is still increasing exponentially. As Dennard scaling [24, 25] came to an end, power density became a problem as more transistors are crammed together. The increased power on a tiny area leads to higher heat generation than conventional cooling solutions can dissipate. To address this issue, researchers have been working towards achieving higher performance without a further increase in power density. Fig. 1.2 shows the hot spot (heavily used parts of the cores) power density at each technology node. The figure shows that hot spot power density can increase by up to 3× from 45nm to 10nm.

While improving performance, one must take care not to exceed the physical limitation of power dissipation. Thus, energy efficiency is key to achieve additional performance gain. Moreover, processors targeting laptops, smartphones and other battery operated mobile devices have always been energy-constrained. In such cases, improved energy efficiency would allow for longer battery life. More recently, mobile processors have become increasingly popular in alternative domains, such as high performance computing due to their low energy budget [26, 27]. Building data centers from low-cost embedded processors is believed to have a huge potential and could change the landscape of supercomputing in the future [27].

1.1 Runtime Energy Management

Traditionally, software applications have been written for serial computation that are executed on a single-core processor. To improve the performance of such applications, techniques such as Instruction-Level Parallelism (ILP), simultaneous execution of multiple instructions with no data dependency; and Memory-Level Parallelism (MLP), servicing multiple outstanding memory accesses at a time), have been used. However, due to the limited ILP and MLP, the single core processors are not effective in improving the performance [2]. Prior art showed that parallelism inherent to an application, including thread-/task-level parallelism (TLP) and data-level parallelism (DLP), is a power-efficient way of achieving high performance than frequency scaling under power limitations. Considering these factors and increased power density, the modern embedded systems [28–30] are employing multi-core processors to utilize the billions of transistors efficiently. Moreover, the energy efficiency has been a primary design objective in embedded systems [31]. These multi/many-core (expected to have 1000+ cores on a single chip [32]) architectures have been demonstrated as a promising solution for energy-efficient computing [33, 34].

Fundamental to improving energy efficiency is managing the operation of the processor in an intelligent way. Modern processors employ various power-saving techniques, such as Dynamic Voltage and Frequency Scaling (DVFS), clock gating, power gating and multiple asymmetric cores (e.g., ARM's big.LITTLE^{TM}architecture [4]). DVFS is one of the promising solutions for energy minimization under performance constraints and workload variations. It has been proved that runtime energy management software can make significant energy improvements by controlling these power-saving techniques to smartly manage the energy-performance trade-off while taking other factors into account (e.g., performance requirements and workload variations) [20, 35]. DVFS is usually controlled by such software, e.g., Linux's power governors and ARM's Voltage and Frequency Scaling (VFS) firmware [36]. The basic principle of DVFS is to reduce the operating voltage-frequency (*V-f*) dynamically at runtime, resulting in a cubic decrease in power consumption [37–39]. A runtime manager can also manage the allocation of processing cores to an application by taking performance constraints, available resources, and optimization goal into account. This is usually called task mapping or thread-to-core mapping, which defines the number of cores and their type allocated to each application. Previous studies [19, 40] showed that efficient thread-to-core mapping leads to energy savings and better performance.

1.2 Research Justification

The existing approaches for runtime energy management for multi-core platforms are generally of three types: 1) offline optimization, 2) online optimization, 3) hybrid (online

optimization is supported by offline profiling). The existing offline optimization based approaches employ DVFS and/or task mapping to find performance-energy trade-off points [4, 41–46]. However, these works have several drawbacks, such as they consider a single application at a time [41–45], and most of them are not evaluated on real hardware platforms [42, 45, 46].

Online optimization has also been considered to cater for dynamic workload scenarios to optimize energy consumption while respecting timing constraints [20, 35, 47–49]. In [47–49], the online algorithm uses hardware performance monitoring counters (PMCs) to achieve energy savings without recompiling the applications. In [20, 35], an online reinforcement learning based DVFS is proposed. These approaches can handle unknown applications, but lead to inefficient results as optimization decisions need to be taken quickly, and offline analysis results are not used. Further, they are agnostic of workload variations of concurrently executing applications. Note that, in this thesis, concurrent execution refers to simultaneous execution of multiple applications that may support various types of parallel computation such as ILP, MLP, DLP, and TLP.

There has also been a focus on online optimization facilitated by offline analysis results [5, 10, 19, 40, 50–52]. Such approaches lead to better performance results than only online optimizations as they take advantage from both offline and online computations. In [40], thread-to-core mapping and DVFS is performed based on power constraint. Similarly, in [5], first thread-to-core mapping is obtained based on utilization, and then DVFS is applied depending upon the surplus power. However, the approaches of [5, 40] have several drawbacks, such as they target homogeneous multi-core architectures, and do not take the workload variations into account while setting the *V-f*.

For heterogeneous multi-cores, whose processing cores have distinct micro-architecture with the same (ARM's big.LITTLE architecture) or different instruction-set architecture (ARM's big.LITTLE CPU + Mali GPU), recent works have considered multi-threaded applications [10, 19, 50–52]. However, at a given moment of time, most of these approaches map application threads completely onto one type of core situated within a cluster [10, 19, 51, 52]. This reduces the thread-to-core mapping complexity, but misses to benefit from the distribution of thread of an application to multiple types of cores. Unlike [10] and [51], the approaches of [19, 50] exploit DVFS, but they have several drawbacks. For example, in [19], each single-threaded application is mapped onto only one type of core at a time. In [50], application threads are mapped onto more than one type of cores, but it heavily depends on offline regression analysis and DVFS level is not adjusted during execution. Moreover, the above approaches do not perform adaptations (changing DVFS setting and/or thread-to-core mappings) upon an application arrival or completion. As demonstrated in this thesis, it leads to lower resource utilisation, increased energy consumption and frequent violation of performance constraints.

Multi-core systems usually execute multiple applications with distinct performance constraints concurrently. The interference between applications due to shared resources, workload variations, and limited and dynamic resource availability make runtime energy management a challenging task. As existing approaches are not effective for concurrent execution of multiple applications, there is a need for efficient runtime energy minimization approaches to address the above problems, which is the aim of this thesis.

1.3 Research Questions

The above discussion motivates us to answer the following research questions related runtime energy management:

1. Can the application workload be classified efficiently at runtime? Subsequently, how can workload classification be applied to concurrent execution of applications on homogeneous multi-cores to achieve energy efficiency with negligible performance loss?
2. For concurrently executing multi-threaded applications on a heterogeneous multi-core system, how can available heterogeneity and the DVFS potential of cores be efficiently exploited to minimize energy consumption while satisfying performance and resource constraints?
3. How can the dependency of runtime decisions on application offline profiling be removed to facilitate energy-efficient management of unknown applications and adaptation to runtime execution scenarios (e.g., application arrival/completion, performance violations, etc.)?

1.4 Research Contributions

To address the above research questions, the contributions that have been made during the course of this research are summarized as follows:

1. **A runtime energy minimization approach through concurrent workload classification on homogeneous multi-cores**. To select appropriate DVFS settings as per the workload, an online workload classification is proposed by taking concurrent execution and contention on shared resources into account. The DVFS settings are decided proactively through workload prediction, thereby adapting to workload variations efficiently and reducing power consumption. The online concurrent workload classification is done using a metric called memory reads per instruction (MRPI), derived through extensive statistical analysis of PMC data and

identifying appropriate PMCs that can capture workload variations at runtime. Further, this approach has been extended to utilise the thread synchronisation overheads, non-uniform memory access latencies and underlying DVFS architecture to improve energy efficiency for multi-threaded applications. It proposes a binning based approach for efficiently determining the DVFS setting as per the predicted workload. Validation on three hardware platforms (Odroid-XU3, Intel Xeon E5-2630, and Xeon Phi 7620P) approach achieves an average improvement in energy efficiency of up to 60% compared to existing approaches. Above contribution addresses Research Question 1, and has been published at IEEE/ACM DATE-2018 [53], AMAS Workshop-2018 [54] and IEEE OPTIM-2018 [55].

2. **An RTM approach for minimizing energy consumption of concurrently executing multi-threaded applications on a heterogeneous multi-core.** This approach first selects thread-to-core mapping based on the performance requirements and resource availability. Unlike, existing approaches, the thread-to-core mapping simultaneously exploits different types of cores for each performance-constrained application in a concurrent execution scenario, thereby improving energy efficiency and performance. Then, it applies online adaptation by adjusting the *V-f* levels of each cluster through concurrent workload classification to achieve energy optimization, without trading-off application performance. Here, the DVFS approach proposed for homogeneous multi-cores in Contribution 1 is extended to take core heterogeneity into account. The experimental results show that the proposed approach achieves an average improvement of up to 33% in energy efficiency compared to existing approaches while meeting the performance requirements. This contribution addresses Research Question 2, and has been published in IEEE/ACM DATE-2017 [56] and IEEE TMSCS-2017 [57].

3. **An energy-efficient adaptive mapping and DVFS approach.** The key feature of the proposed approach is the elimination of dependency on offline profiled results while making runtime decisions. This is achieved through a performance prediction model having a maximum error of 7.9% lower than the previously reported model and an energy-efficient mapping approach that allocates processing cores to applications while respecting performance constraints. Furthermore, this approach adapts to runtime execution scenarios efficiently by monitoring the application status, and performance/workload variations to adjust the previous DVFS settings and thread-to-core mappings. The proposed approach is experimentally validated on the Odroid-XU3, with various combinations of diverse multi-threaded applications from PARSEC and SPLASH benchmarks. Results show an improvement of up to 28% in energy efficiency compared to the recently proposed approach while meeting performance constraints. This contribution addresses Research Question 3, and has been submitted for publication in IEEE TCAD-2019 [58].

In the co-authored publications below, that have resulted from the research work presented in this thesis, my contributions include developing the ideas, implementation and validation of the approach, and writing most of the papers. The remaining co-authors have been primarily involved in discussions on how best to carry out the experimental validations and place the research contributions in the context of relevant previous work and/or providing feedback to improve the presentation of the paper.

1.5 Software Contributions

Intelligent Runtime Management Algorithms: The RTM algorithms presented in Chapter 3 have been implemented as userspace applications for experimental validation. The source codes are available at the addresses: https://github.com/PRiME-project/rtm-Xeon (Xeon platforms) and https://github.com/PRiME-project/RTM_Concurrent (Odroid-XU3). This RTM has been further integrated into the cross-layer framework, the PRiME Framework (https://github.com/PRiME-project/PRiME_Framework), for experimental validation.

Furthermore, the runtime energy management approaches presented in Chapter 4 and 5 are also implemented as userspace applications, whose source code is made available at the address: https://github.com/PRiME-project/RTM_Concurrent and https://github.com/PRiME-project/rtm-dtcm, respectively.

1.6 Publications

The following publications have no PhD student as a co-authors and therefore, they have not been reported in anyone else's thesis.

1. **K. R. Basireddy**, A. K. Singh, G. V. Merrett, and B. M. Al-Hashimi, "AdaMD: Adaptive Mapping and DVFS on Heterogeneous Multi-cores," *submitted* to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2019 [submitted].

2. **K. R. Basireddy**, A. K. Singh, G. V. Merrett, B. M. Al-Hashimi, "Online concurrent workload classification for multi-core energy management," Design, Automation and Test in Europe (DATE), 2018. [53]

3. **K. R. Basireddy**, E. W. Wachter, G. V. Merrett and B. M. Al-Hashimi, "Workload-Aware Runtime Energy Management for HPC Systems," in IEEE International Workshop on Optimization of Energy Efficient HPC & Distributed Systems (OPTIM), 2018. [55]

4. **K. R. Basireddy**, E. W. Wachter, B. M. Al-Hashimi and G. V. Merrett, "Memory- and thread synchronization contention-aware DVFS for HPC systems," Adaptive Many-Core Architectures and Systems workshop, 2018. [54]

5. **K. R. Basireddy**, A. K. Singh, G. V. Merrett, and B. M. Al-Hashimi, "ITMD: Run-time Management of Concurrent Mulit-threaded applications on Heterogeneous Multi-cores," Design, Automation and Test in Europe (DATE), pp. 1, 2017. [56]

6. **K. R. Basireddy**, A. K. Singh, Dwaipayan Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster Thread-to-core Mapping and DVFS on Heterogeneous Multi-cores," IEEE Transaction on Multi-Scale Computing Systems (TMSCS), 2017. [57]

As part of the EPSRC funded PRiME project, the following publications were also made through collaboration, which are not reported as contributions in this thesis:

1. A. K. Singh, **K. R. Basireddy**, A. Prakash, G. V. Merrett, and B. M. Al-Hashimi , "Energy-Efficient Collaborative Adaptation in Heterogeneous Mobile SoCs," IEEE Transactions on Computers (TC), 2018 [59] (*under review*).

2. E. W. Wachter, C. d. Bellefroid, **K. R. Basireddy**, A. K. Singh, B. M. Al-Hashimi, and G. V. Merrett, "Predictive Thermal Management for Energy-efficient Execution of Concurrent Applications on Heterogeneous Multi-cores," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019 [60].

3. A. K. Singh, A. Prakash, **K. R. Basireddy**, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs," ACM Transactions on Embedded Computing Systems (TECS), 2017. [61]

4. A. K. Singh, C. Leech, **K. R. Basireddy**, B. M. Al-Hashimi, and G. V. Merrett, "Learning-based run-time power and energy management of multi/many-core systems: current and future trends," Journal of Low Power Electronics (JOLPE), 2017. [62]

5. **K. R. Basireddy**, M. J. Walker, D. Balsamo, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Empirical CPU power modelling and estimation in the gem5 simulator,âĂİ in IEEE International Conference on Power and Timing Optimization and Simulation (PATMOS), 2017. [63]

1.7 Thesis Organization

The remainder of the thesis is organized as follows.

Chapter 2 first presents an introduction to energy consumption, DVFS and thread-to-core mapping. It then reviews the existing runtime energy management systems and identifies the drawbacks in those approaches.

Chapter 3 presents the investigation on workload classification using execution time and hardware PMC data. Further, it discusses the proposed energy minimization approach for concurrently executing single-threaded applications, specifically on a homogeneous multi-core system supporting cluster-wide DVFS (cores in a cluster are set to the same DVFS level). It also presents the runtime classification of concurrent workloads and selection of appropriate DVFS level based on the predicted workload.

Chapter 4 extends the online DVFS technique proposed in Chapter 3 to a cluster-based heterogeneous multi-core architecture. It also proposes inter-cluster thread-to-core mapping to efficiently manage multi-threaded applications to improve energy efficiency while meeting their performance and system resource constraints.

Chapter 5 presents an energy efficient adaptive mapping approach coupled with DVFS for performance-constrained applications, executing on heterogeneous multi-cores. It starts with an introduction and gives a motivational example to show the importance of the proposed approach. Then, a detailed discussion on problem formulation and various steps in the proposed approach are given. Finally, experimentally evaluation of performance models and the runtime management approach are carried out to show the benefits in comparison to the existing approaches.

Chapter 6 concludes this work, providing a summary of the key contributions found throughout this thesis. Additionally, potential future directions as an extension are detailed.

Chapter 2

Literature Review of Runtime Energy Management

This chapter presents a necessary background and a thorough review of existing approaches to runtime energy management. Embedded systems, which are mainly referred to as a fixed or programmable system for executing a specific function, have crossed the boundaries of embedded computing by supporting general purpose computing with sophisticated system software. For example, mobile phones used to be part of a cellular network, but they now serve various purposes such as audio/video player, video games, voice calling, etc. Similarly, real-time embedded systems that exist in self-driving cars have sophisticated computing infrastructure to handle various aspects (e.g., constructing internal maps, processing input and plotting the path, etc.) As a result, these systems have been employing multi-core processors for meeting the ever-rising performance and power demands of modern complex embedded applications and for improving the energy efficiency. Multi-core platforms containing several homogeneous and/or heterogeneous processing cores are becoming ubiquitous in embedded processing [64]. These platforms can provide better performance-energy tradeoffs by executing parallel threads of applications on different types of processing cores simultaneously. In addition to that, in some cases, each processing core could operate at a different Dynamic Voltage and Frequency Scaling (DVFS) level, possibly at a lower frequency unlike at a very high frequency in single-core processors and thus fulfilling the energy consumption constraint.

In this context, runtime energy management is required to adapt to diverse resource and performance requirements of applications and to achieve energy savings. There are several approaches proposed for homogeneous and heterogeneous multi-core embedded platforms to improve energy efficiency. These approaches usually consider DVFS and/or thread-to-core mapping techniques for efficiently managing the available system resources. For a better understanding of the existing and proposed runtime energy management approaches, DVFS and thread-to-core mapping concepts are briefly explained

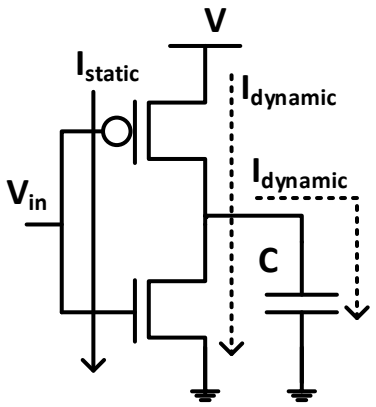

FIGURE 2.1: Illustration of static and dynamic current through an inverter gate

in Section 2.1. The recent architectural innovations to improve performance and energy efficiency are given in Section 2.2. Then, a critical review of existing runtime energy management approaches is presented in Section 2.3. This section mainly covers runtime energy management approaches for homogeneous and single-Instruction Set Architecture (ISA) heterogeneous architectures, as these are the focus of this thesis. Finally, the contents of this chapter are summarized in Section 2.5.

2.1 DVFS and Thread-to-Core Mapping

The two components that determine total energy consumption in a system are static and dynamic/switching energy consumption. Static energy consumption is caused by a small current continuously leaking through the transistors, while dynamic is due to charging and discharging of output capacitance [65]. The total energy consumption (E_T) can be represented as a product of total power consumption (P_T) and execution time (t_e) as follows:

$$E_T = P_T * t_e \tag{2.1}$$

where P_T is computed as below:

$$P_T = P_{dynamic} + P_{static} \tag{2.2}$$

Figure 2.1 shows how current flows through an inverter gate at the transistor level. The solid arrow indicates where static leakage occurs, and the two dotted arrows show where charges escape during switching. Static power consumption origins from transistor size and layout, while dynamic power consumption depends on the amount of transistor switching.

Energy consumption of large-scale and embedded computing systems has been increasing by over 15% per year [66]. Furthermore, due to the growing need for high performance yet energy-efficient computing systems, energy management has central importance almost in every current and future computing domain [67]. As a consequence, modern embedded systems are equipped with various low power techniques such as power gating, clock gating, and DVFS. To achieve energy efficiency, the aforementioned techniques need to be managed in an intelligent way during the system operation. Clock gating and power gating are two widely used techniques to reduce dynamic/leakage power by shutting off the unused components and cores from clock tree and power supply. For runtime adaptation and achieving energy efficiency, DVFS helps to scale the Voltage-frequency (V-f) during application execution. Moreover, thread-to-core mapping helps in allocating the system resources (number of cores and their type) to each application to minimize energy consumption while meeting performance and resource constraints.

2.1.1 DVFS

DVFS is a commonly-used technique to save dynamic power, under the workload and/or performance variations, on a wide range of computing systems, from embedded, laptop and desktop computing systems to high-performance server-class systems [68]. DVFS has been used as a means to improve power efficiency by lowering the CPU frequency and voltage when cycles are being wasted, stalled on memory resources, or under low performance requirements. It is based on the following relation between frequency, voltage, and power:

$$P_{dynamic} \propto C * f * V^2 \tag{2.3}$$

where C, f and V represent load capacitance, operating frequency and supply voltage, respectively. DVFS takes advantage of the quadratic relationship between supply voltage and energy consumption [69], which can result in significant energy savings. Fig. 2.2 shows the maximum operating frequency and dynamic power dissipation of one core versus supply voltage. However, DVFS techniques pose a difficult challenge to real-time systems as it has to minimize the energy without impacting the desired quality of service offered by the system to applications and end users. Moreover, energy can only be saved if the power consumption is reduced enough to cover the extra time it takes to run the workload at a lower frequency.

The basic concept of DVFS for real-time application scenarios is illustrated in Fig. 2.3 [4]. In this figure, T_2 and T_4 denote deadlines for tasks W_1 and W_2, respectively. W_1 finishes at T_1 if the CPU is operated with a supply voltage level of V_1. The CPU will be idle during the remaining (slack) time, S_1. To provide a precise quantitative example, let us assume T_2 - T_0 = T_4 - T_2=ΔT, and T_1 - T_0 =ΔT/2; the CPU clock frequency at V_1 is

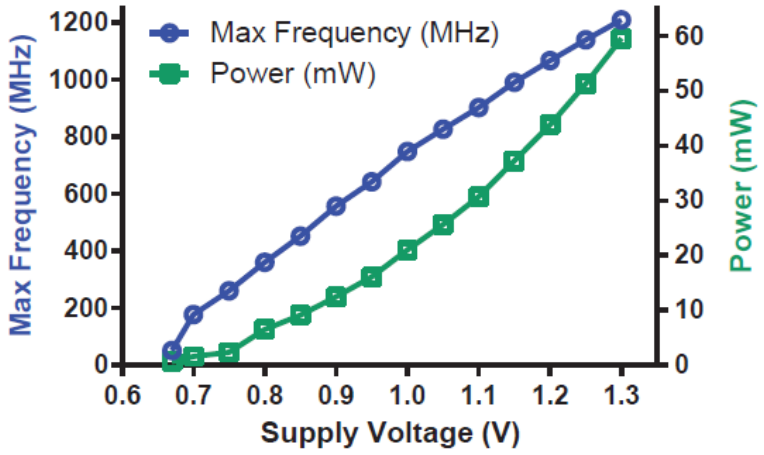

FIGURE 2.2: Relation between maximum frequency and dynamic power versus supply voltage. Reprinted from [3].

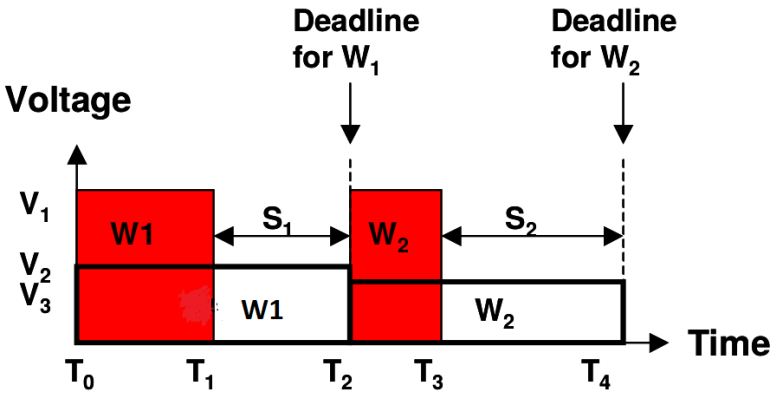

FIGURE 2.3: An illustration of the DVFS technique. Reprinted from [4].

f_1 =n/ΔT for some integer n; and that the CPU is powered down or put into standby with zero power dissipation during the slack time. The total energy consumption of the CPU:

$$E_1 = CV_1{}^2 f_1 \Delta T/2 = nCV_1{}^2/2 \tag{2.4}$$

where C is the effective switched capacitance of the CPU per clock cycle. Alternatively, W_1 may be executed on the CPU by using a voltage level of V_2 =V_1/2, and is thereby completed at T_2. Assuming a first-order linear relationship between the supply voltage level and the CPU clock frequency, f_2 =f_1/2. The total energy consumed by the CPU:

$$E_2 = CV_2{}^2 f_2 \Delta T = nCV_1{}^2/8 \tag{2.5}$$

Clearly, there is a 75 % energy saving ($\frac{(E1-E2)\times 100}{E1}$) as a result of lowering the supply voltage (this saving is achieved in spite of perfect, i.e., immediate and with no overhead,

power down of the CPU). This energy saving is achieved without sacrificing the QoS because the given deadline is met. An energy saving of 89% is achieved when scaling V_1 to $V_3 = V_1/3$ and f_1 to $f_3 = f_1/3$ in case of task W_2.

Weiser *et al.* [70] were the first to propose the use of DVFS to reduce the energy consumption of processors using the simulated execution traces and level of slack time for choosing the new frequency. Weissel and Bellosa [47] developed models for predicting workload's response to a change in frequency based on memory requests per cycle and instructions per cycle, counted using the Performance Monitoring Unit (PMU) available in most processors. Snowdon *et al.* [71] enhanced this approach by developing a technique to automatically choose the best model parameters from the hundreds of possible events that modern PMUs can measure.

Multi-core processors usually support per-core and per-cluster DVFS to improve energy efficiency through independent control of *V-f* levels of each core and cluster, respectively. Homogeneous multi-core architectures can emulate heterogeneity with fine-grained per-core DVFS as individual core power/performance varies with selected *V-f*. For per-cluster DVFS, a group of homogeneous cores is tuned to a particular *V-f* through some intelligent mechanism. DVFS is controlled by a runtime manager to achieve energy efficiency while providing a better quality of service.

2.1.2 Thread-to-Core Mapping

With technological advancement and increasing performance demands, the number of cores in the same chip area has grown exponentially, and different types of cores have been integrated. To efficiently exploit the underlying core-level parallelism, application threads have to be carefully allocated onto the processing cores to minimize the overall energy consumption while satisfying the performance and resource constraints. This process is usually called as thread-to-core mapping, or task mapping, or simply mapping. The process of mapping involves binding of applications to underlying hardware resources, i.e., processing cores. In case of heterogeneous multi-core system, in addition to the allocation of a set of processing cores, the mapping defines the number of processing cores and their type. The mapping and scheduling problem is similar to *Quadratic Assignment Problem*, a well-known NP-hard problem [72]. Therefore, finding an optimal solution satisfying all the given constraints is tedious and time-consuming. Thus, heuristics based on application domain knowledge need to be employed to find a nearly optimal solution. The optimal solutions can be explored by advance design-time analysis and then can be used at runtime. However, the explosion in the number of use-cases (combination of simultaneously active applications) with the increasing number of applications makes the analysis infeasible. For n applications, the analysis needs to be performed for 2^n use-cases. Additionally, such an analysis cannot deal with dynamic scenarios such as runtime changing standards and the addition of new applications.

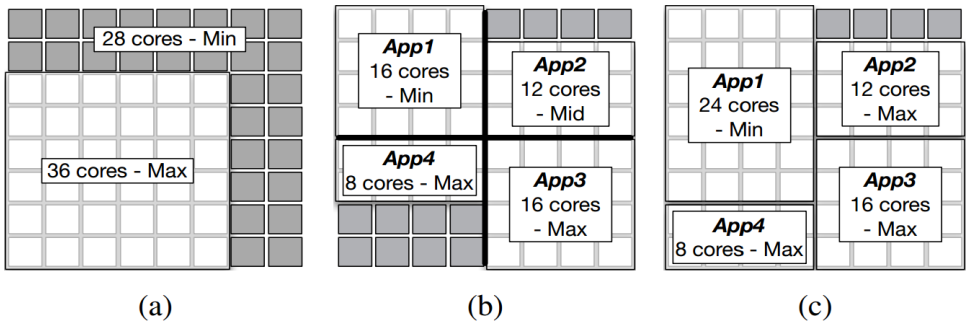

FIGURE 2.4: Illustration of thread-to-core mapping under a power cap. (a) Executing a multithreaded program. (b) and (c) Executing four multithreaded applications concurrently. Reprinted from [5].

Runtime management is required to handle such dynamism albeit optimal mapping solutions are not found. The application mapping problem has been identified as one of the most urgent problems to be solved for implementing embedded systems [73, 74].

Fig. 2.4 (a) shows an example of thread-to-core mapping through thread packing (a process of confining application threads onto a variable number of cores) and DVFS applied to a 64-core processor running a multi-threaded program [5]. The resource distribution is optimized by allocating 36 cores with maximum frequency while keeping the rest of the cores idle with minimum frequency. The allocation becomes much more complicated when multiple multi-threaded applications are executed concurrently. A straight forward approach is to equally partition the resources (cores and power budget) between the applications and apply an optimization within each partition. Fig. 2.4 (b) shows such an example with the execution of four programs, App1, App2, App3 and App4, optimized by allocating 16 cores with minimum frequency (or 16-Min), 12-Mid, 16-Max, and 8-Max, respectively. As the performance is locally optimized for each partition, this is not always guaranteed to be the globally optimal assignment. For example, consider a situation where App3 and App4 have more power budget than are sufficient; App3 is fully utilizing the partition with maximum frequency but there is still surplus power (i.e., not power hungry) and App4's performance cannot be improved by adding extra cores (i.e., poorly scalable). Where on the other hand, App1 and App2 have the potential to increase their performance if more power is allowed to be consumed. Fig. 2.4(c) illustrates an example where the performance is further improved than that of Fig. 2.4(b) by allowing the surplus power of App3 and App4 to be consumed by App1 and App2 by increasing the number of cores assigned (App1) and by operating at a higher frequency (App2). The exploration space for such global optimization is multi-dimensional where the allocation of the number of cores and the operating frequency of each program can be varied under two global conditions: sum of the allocated number of cores must be less than or equal to the available number of cores, and the power consumption must satisfy the constraint.

For static and dynamic workload scenarios, the mapping methodologies perform optimization at design-time and runtime respectively, which has led them to classify as design-time and runtime methodologies, respectively - these methodologies target either homogeneous or heterogeneous multi-core systems. The runtime mapping requires a platform manager that handles mapping of threads at runtime. In addition to mapping, the manager is also responsible for thread scheduling, resource control and thread migration at runtime [75].

Design-time mapping methodologies are suitable for static workload scenarios where a predefined set of applications with known computation and communication behavior and a static platform are considered [76]. They are unable to handle dynamism in applications incurred at runtime (e.g., multimedia and networking applications). Examples of such dynamism could be adding a new application into the system at runtime. Since applications are often added to the platform at runtime (for example, downloading a Java application in a mobile-phone at runtime), workload variations take place. So, runtime mapping methodologies have to handle such dynamic workloads. The runtime mapping methodologies face the challenge to map tasks of new applications on the platform resources to satisfy their performance requirements while keeping accurate knowledge of resource availability. After mapping threads, thread migration can also be used to revise placement of some of the already executing applications if the performance requirement changes or a new application is added to the system.

2.2 Multi-Processor System-on-Chips

This section describes various architectural innovations that have been proposed over the years to address the ever-increasing performance and/or power demands of modern embedded Multi-Processor System-on-Chip (MPSoC) platforms.

2.2.1 Number of Processing Cores

The advancements in process technology helped to double the density of transistors, as predicted by Moore's law. This trend has continued even in the number of cores with roughly the same exponential rate according to International Technology Roadmap for Semiconductors (ITRS), which says that this growing trend will continue as shown in Fig. 2.5. Thus, as process technology evolves, integration of thousands of cores - just like logic gates - on the same chip becomes a reality as predicted by sources like Intel and Berkeley [32, 77]. As a result, almost all computing vendors, such as Intel, AMD, etc., have been producing processors with an increasing number of cores, approximately doubling the number of cores per chip per year. These chips have revolutionized the processor architecture and are widely called as chip multiprocessors, multi-core chips,

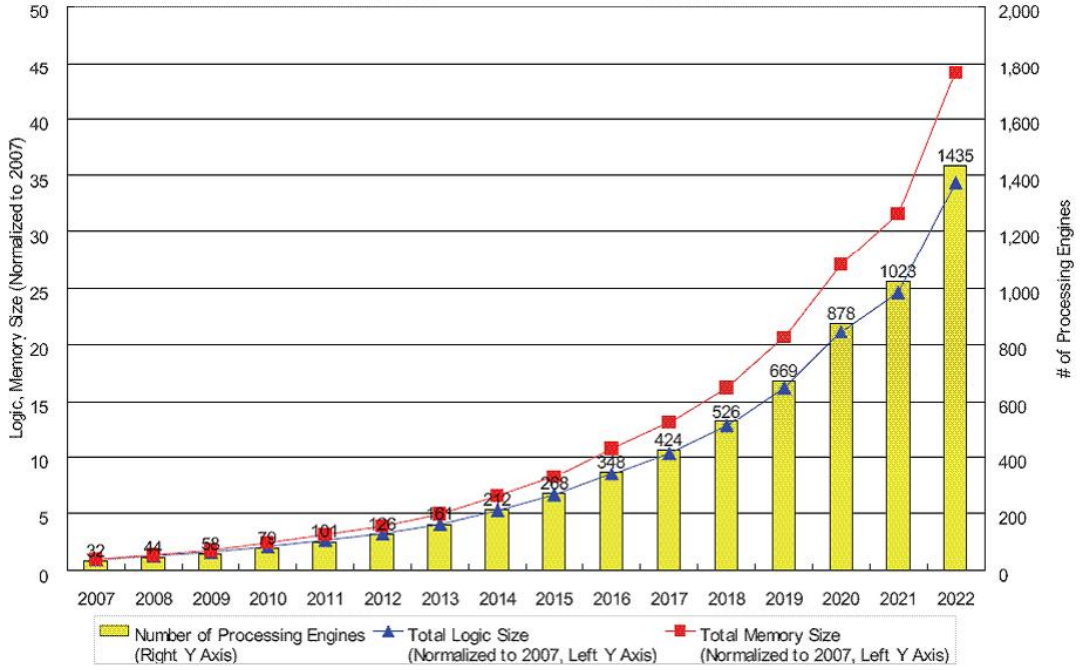

FIGURE 2.5: Evolution of the number of cores in embedded platforms. Extracted from [6].

and many-core chips. The complete system employing such processors is commonly referred to as MPSoC.

2.2.2 Heterogeneity in Processing Cores

Multi-threading/parallel processing of an application has made possible to see the true benefits of having more processing cores in a multi-/many-core era. According to Amdahl's law [78], speedup of an application through parallel execution on a multi-core processor is limited by the time needed to execute the sequential section of the application. As per the Amdahl's law, theoretical speedup (S) of the execution of the whole task is formulated in the following way:

$$S = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.6}$$

where, s is the speedup of the part of the task that benefits from improved system resources and p is the proportion of execution time that the part benefiting from improved resources originally occupied. Fig. 2.6 shows the speedup obtained by using the different number of processors at various levels of parallelization. It is clear that if 5% of the application cannot be parallelized (95% parallelized) then the maximum speedup that can be achieved is 20× and it cannot be improved further even by increasing the number of processors. However, the speedup can be increased by accelerating the execution of the non-parallelized part (5%), which can be facilitated by executing the sequential part on a

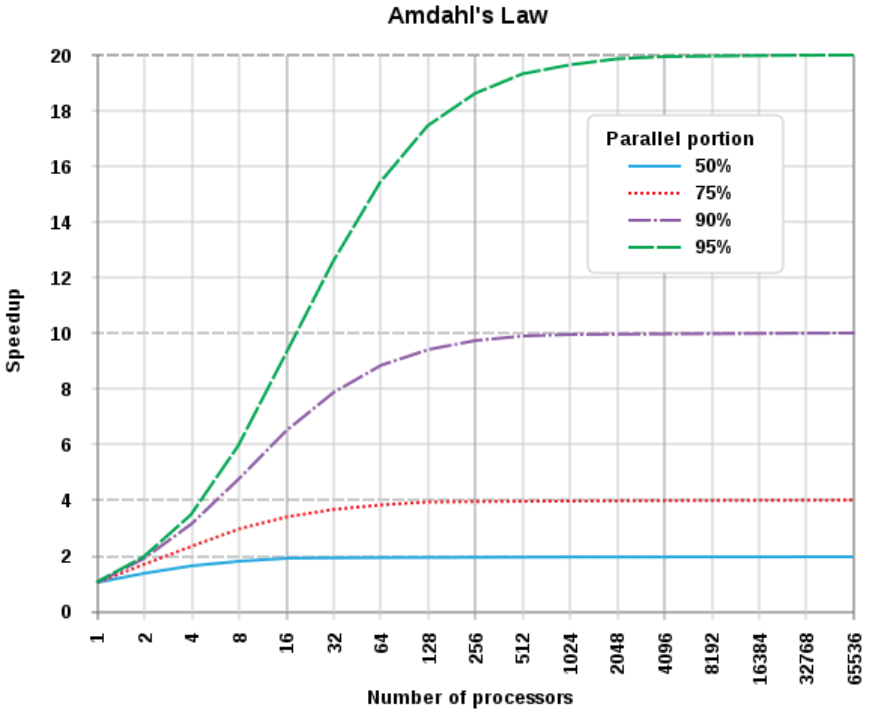

FIGURE 2.6: Speedup vs. number of processors for different portions of parallel code in an application, using Amdahl's law [7].

powerful processor with a better sequential performance. This can be achieved through heterogeneous multi-core processors, where various portions of the application can exploit the distinct features of different types of cores. Thus, heterogeneous MPSoCs have become promising computing alternatives to achieve significant performance gains over their homogeneous counterpart. Further, heterogeneous MPSoCs may contain general purpose processor (GPP) for flexibility, custom accelerators for compute-intensive tasks, reconfigurable hardware blocks for flexibility, and specialized processors like digital signal processors (DSPs) for signal processing tasks. Such system architecture could also help reduce power consumption and improve performance simultaneously. This would also provide an opportunity to meet the high-performance demands of complex applications and to migrate tasks onto the appropriate type of cores when applications go through different phases of execution.

2.2.3 Types of Multi-core Architecture

Implementation of processors using multi-core architecture has proved to suitable to most of the application domains [64]. The multi-core architecture offers a number of advantages over single-core architectures, which are listed below [79]:

1. High transistor density offered by future process technologies can be well utilized by having more cores while keeping the complexity of the cores the same.

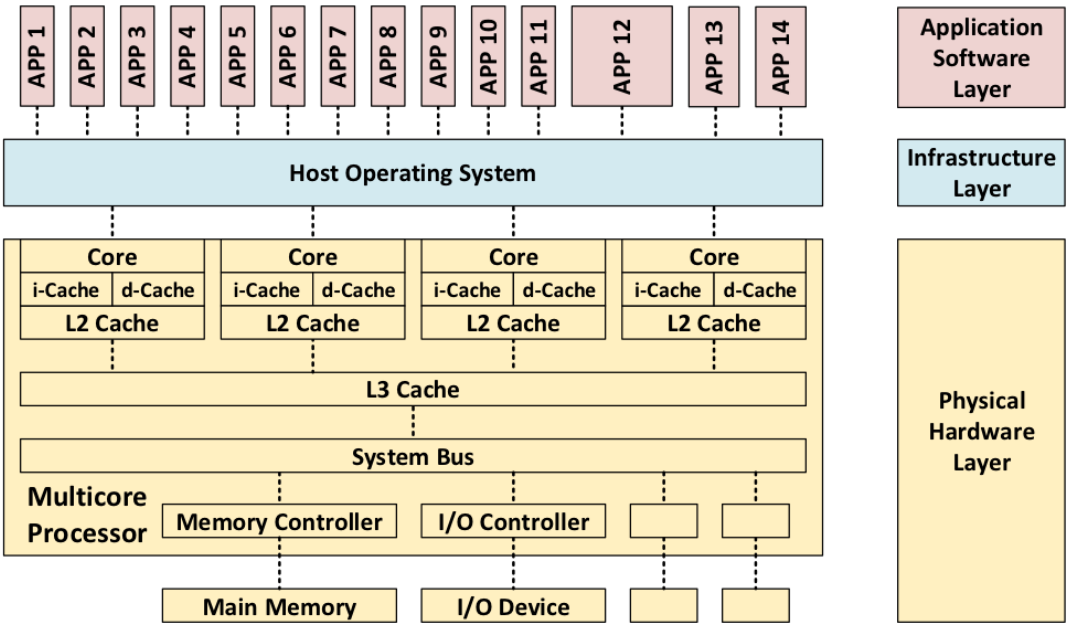

FIGURE 2.7: Homogeneous multi-core architecture of a system in which 14 software applications are allocated by a single host operating system to the cores, reprinted from [8].

2. Small cores with less complex architecture can be optimized extensively.
3. Computational performance can be improved with the number of cores.
4. Some cores can be switched off/on depending on the requirements.
5. Faulty cores can be discarded to make the multi-core concept as fault tolerant, and multiple cores can be configured in parallel to improve the performance.
6. Individual clock domain per core is viable, and it is possible to do partial dynamic reconfiguration on a per-core basis for the reconfigurable cores.

Several multi-core chips have been developed, targeting different application domains [80]. These multi-core chip platforms are viable alternatives to high-performance computing platforms.

2.2.3.1 Homogeneous Multi-core Architectures

Homogeneous multi-core architectures contain identical cores, having the same power consumption and performance if process variations are neglected. Unlike heterogeneous multi-core architectures, VLSI implementation of homogeneous architectures and programming are generally simple due to the same instruction set architecture (ISA), micro-architecture used for all processing cores, memory hierarchy and Input/Output(I/O) facilities. Typically, each core has a private I-cache, D-cache and L2-cache (in some implementation if there is a shared L3-cache (last-level cache)). The memory subsystem is

connected to I/O and external components through a system bus or other interconnect. The uniform architecture of homogeneous multi-cores makes implementation of system software and application development simple by supporting a single Operating System (OS) that spans all the cores. Such a typical homogeneous quad-core architecture executing 14 software applications using a single operating system is shown in Fig. 2.7 (extracted from [8]).

Academic researchers often propose homogeneous MPSoCs and some of them ones are listed here as an example. A 16-core Raw Architecture Workstation (RAW) processor architecture was proposed by a research group from the Massachusetts Institute of Technology [81]. The University of California at Davis presented a 167-core Asynchronous Array of Simple Processors (AsAP) [82]. The University of Texas at Austin proposes The Tera-op, Reliable, Intelligently adaptive Processing System (TRIPS) with 32 chips each containing two cores [83]. A scalable MPSoC for next-generation architectures has been proposed in [84], which is based on RISC processors and distributed memories. A WaveScalar processor has been proposed by the University of Washington which contains approximately 2K simple processing elements (PEs) arranged into 16 clusters [85]. The common goal all these chips is to offer a high performance to complex applications.

In addition to the academic multi-core chips, commercial chip/IP vendors like Intel, AMD, Arm, and Tilera Corporation also released homogeneous MPSoCs [33, 86]. The homogeneous MPSoC proposed by Intel in [33] contains 80 cores, having two floating-point units, connected by an interconnection network. The interconnection network is arranged as a 10×8 2D array in 275mm^2 area that contains 80 cores and packet-switched routers, operating at 4 GHz. Continuing this trend, Intel has also announced Core i3, Core i5, and Core i7, a family of multi-core processors for desktop and mobile processors [80]. Tilera Corporation announced TILE-Gx100, the world's first 100-core general purpose processor, offering the highest performance amongst any microprocessor at the time announced and simplified many-core programming through Multicore Development Environment (MDE). HP also announced an MPSoC platform containing multiple MIPS cores [87]. This trend has influenced other chip vendors to produce their multi-core platforms targeting different application domains such as scientific, embedded and general purpose computing [80].

2.2.3.2 Heterogeneous Multi-core Architectures

Heterogeneous MPSoCs contain processing cores of different types, having diverse power-performance tradeoffs. The processing elements can be GPPs, processing cores running at different DVFS levels, specialized processors like DSPs, Graphics Processing Units (GPUs), FPGA fabrics, etc. Fig. 2.8 shows a hypothetical hardware and software architecture of a Heterogeneous MPSoC, running four different operating systems and 14 applications [8]. In contrast to the architecture presented in Fig. 2.7, the heterogeneous

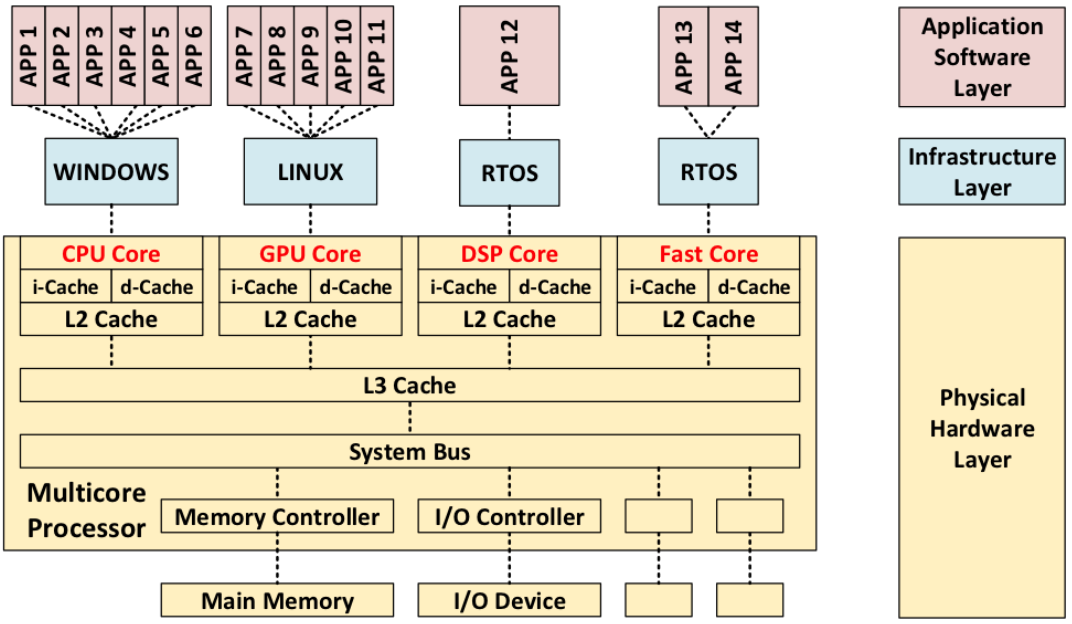

FIGURE 2.8: A heterogeneous multi-core architecture, running four different operating systems and 14 applications, reprinted from [8].

MPSoC shown in Fig. 2.8 has different types of cores with varying ISA and micro-architecture, and simultaneously running OSs that are well suited for a particular type of processing core(s). By carefully exploiting distinct features of the different type of processing cores for an appropriate application, the performance and power consumption can be substantially improved using such an architecture. These advantages have been noticed by academic researchers, who have already proposed various heterogeneous architectures for MPSoCs [88–92].

The processing cores of heterogeneous multi-core architectures are usually designed with varying sizes and complexities to compliment each other in terms of performance and energy efficiency. A general system contains smaller cores for processing simple tasks in an energy efficient way and larger cores for handling compute-intensive tasks with higher performance at the cost of increased power consumption. Further, these heterogeneous cores can have a similar type of Instruction Set Architecture (single-ISA), e.g., ARM big.LITTLE architecture or different (functional heterogeneous), e.g., ARM big.LITTLE with Mali GPU [28]. Single-ISA heterogeneous multi-core processors are typically composed of small (e.g., in-order) power-efficient cores and big (e.g., out-of-order) high-performance cores. There have been various complex MPSoCs proposed, for example, Nollet *et al.* [88] presented an MPSoC with four different types of processing cores: GPPs, DSPs, specialized accelerators, and reconfigurable hardware blocks. Similarly, Arpinen *et al.* [91] proposed an MPSoC consisting of multiple Altera Nios II soft-core processors and custom hardware accelerators that are connected by a communication network called Heterogeneous IP Block Interconnection (HIBI). These systems provide high performance for a fixed set of applications.

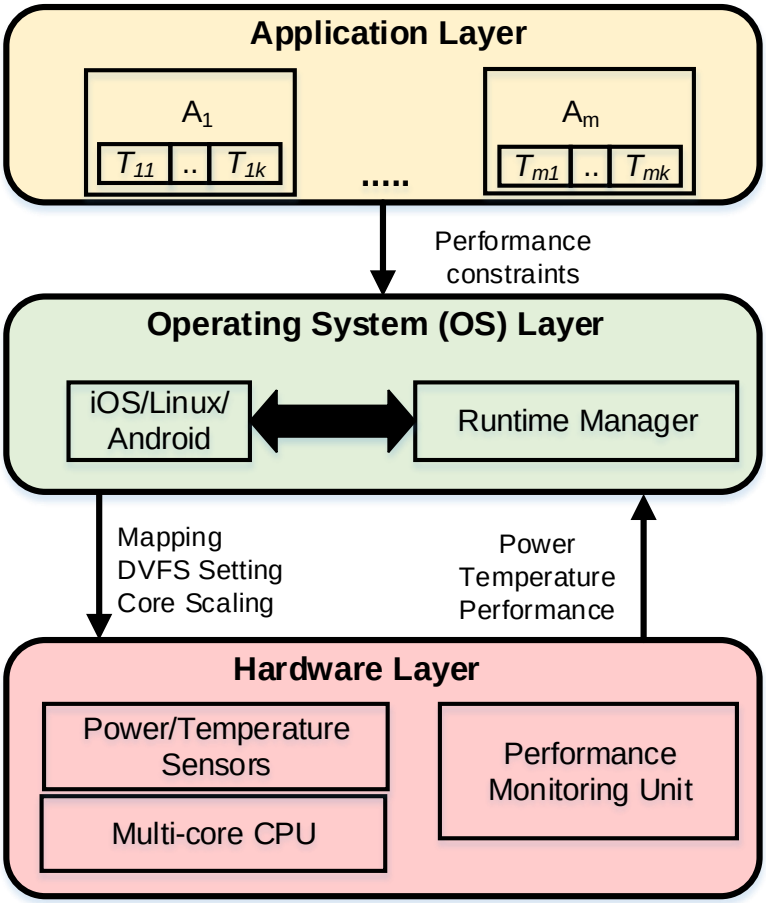

FIGURE 2.9: Representing an embedded systems as three stacked layers: application, operating system and the hardware.

These architectures also support runtime monitoring of various performance statistics through a performance monitoring unit (PMU). For example, the ARM Cortex-A15 includes logic to gather various statistics on the operation of the processor and memory system during runtime, based on the PMUv2 architecture [93]. the PMU usually contains multiple counters (registers), called performance monitoring counters (PMCs). Each PMC can be configured to monitor a specific event (e.g., instructions retired, L1-cache misses, etc). For example, ARM Cortex-A15 [93] and Cortex-A7 [94] processors in the Samsung Exynos 5422 [28], have seven and five 32-bit counters (including cycle counter), respectively. These events provide useful information about the behavior of the processor that can be used for debugging, profiling and making appropriate decisions (mapping and DVFS) during runtime management. They also support real-time measurement of power and temperature, which can be used in design/validation of runtime management system.

2.3 Approaches to Runtime Energy Management

A runtime manager (RTM), in general, is system software that makes runtime intelligent decisions to manage various metrics of the system such as energy, performance, and reliability. Fig. 2.9 shows a three-layer view of a typical embedded system. The topmost layer is the application layer, which is composed of active applications, executing sequentially or concurrently. The middle layer is the operating system layer (e.g., Linux, iOS, Android, etc), which coordinates an application's execution on the hardware. Finally, at the bottom, there is the hardware layer, consisting of multi-core processors with various sensors (e.g., power, temperature) and a performance monitoring unit (PMU). All three layers interact with each other to execute an application, as indicated by arrows in the figure. The RTM is represented by a box inside the operating system layer, which adapts to varying performance requirements and workload through DVFS, core scaling and/or thread-to-core mapping. The runtime manager can be implemented in the user space or kernel space. There have been various runtime energy management systems (RTEM) proposed to improve energy efficiency and meet performance constraints.

Energy minimization in embedded systems is usually achieved through offline optimization, or online optimization (sometimes supported by offline analysis). Based on the underlying hardware architecture, state-of-the-art approaches to runtime energy management from the literature can broadly be classified as follows:

1. Approaches for homogeneous multi-cores
2. Approaches for heterogeneous multi-cores

In addition to the approaches proposed by academic researchers, the Linux kernel offers a set of governors for power management, which are widely used in both homogeneous and heterogeneous mobile and desktop platforms. Therefore, first default Linux power governors are discussed and then, a thorough review of existing approaches is presented.

2.3.1 Linux Power Governors

Linux `cpufreq` offers a set of default power governors for controlling the CPU frequency. Fig. 2.3 shows the high-level `cpufreq` infrastructure, which is taken from [9]. `cpufreq` is the subsystem of the Linux kernel that allows the frequency to be explicitly set on processors [95] and offers a modularized set of interfaces to manage the CPU frequency changes. It primarily contains `cpufreq` module, CPU-specific drivers, in-kernel governors. The `cpufreq` module provides a common interface to the various low-level, CPU-specific frequency control technologies and high-level CPU frequency controlling policies. `cpufreq` decouples the CPU frequency controlling mechanisms and policies and helps in independent development of the two. It also provides some standard interfaces

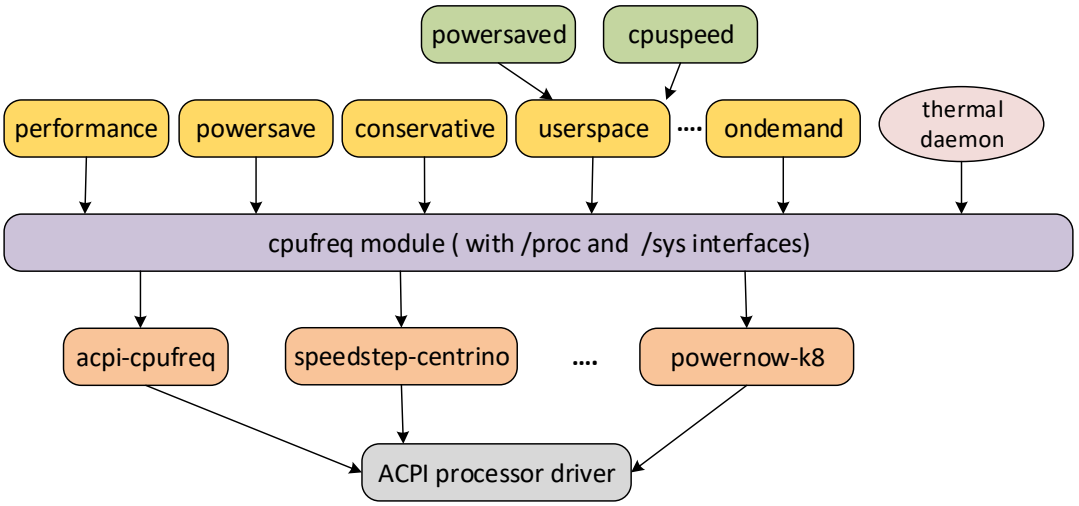

FIGURE 2.10: `cpufreq` infrastructure, a subsystem of the Linux kernel. Extracted from [9].

to the user, with which the user can choose the policy governor and set parameters for that particular policy governor. CPU-specific drivers implement different CPU frequency changing technologies, such as Intel® SpeedStep® Technology, Enhanced Intel® SpeedStep® Technology [6], AMD PowerNow!™, and Intel Pentium® 4 processor clock modulation. On a given platform, one or more frequency modulation technologies can be supported, and a proper driver must be loaded for the platform to perform efficient frequency changes. The `cpufreq` infrastructure allows the use of one CPU-specific driver per platform. Some of these low-level drivers also depend on Advanced Configuration and Power Interface (ACPI) methods to get information from the Basic Input/Output System (BIOS) about the CPU and frequencies it can support. The `cpufreq` infrastructure allows for frequency-changing policy governors, which can change the CPU frequency based on different criteria, such as CPU usage. It can also show available governors on the system and allows the user to select a governor to manage the frequency of each independent CPU. The recent Linux Kernel 3.10.96+ comes bundled with six different governors. The user interface to `cpufreq` is through `sysfs`. cpufreq provides the flexibility to manage CPUs at a per-processor level (as long as hardware agrees to manage CPUs at that level). The interface for each CPU is under `sysfs`, typically at `/sys/devices/system/cpu/cpuX/cpufreq`, where X ranges from 0 through N-1, with N being the total number of logical CPUs in the system.

`Performance` governor sets the CPU statically to the highest frequency within the user-specified range between 'scaling_min_freq' and 'scaling_max_freq'.

`Powersave` governor locks the CPU frequency at the lowest frequency set by the user scaling_min_freq and scaling_max_freq.

`Userspace` governor allows the user, or any userspace program running with 'root' privilege, to set the CPU to a specific frequency by making a `sysfs` file 'scaling_setspeed'

available in the CPU-device directory. All userspace dynamic CPU frequency governors use this governor as their proxy.

`Ondemand` governor sets the CPU frequency depending on the current system load. Load estimation is triggered by the scheduler through the update_util_data→func hook; when triggered, cpufreq checks the CPU-usage statistics over the last period, and the governor sets the CPU accordingly. The CPU must have the capability to switch the frequency very quickly.

`Conservative` governor is much like the `ondemand` governor, and sets the CPU frequency depending on the current usage. It differs in behavior by gracefully increasing and decreasing the CPU speed rather than jumping to max speed the moment there is any load on the CPU. This feature is more useful in a battery powered environment.

`Interactive` governor is designed for latency-sensitive and interactive workloads. The CPU speed is set depending on usage, just like '`ondemand`', but the governor is more aggressive about scaling the CPU speed up in response to CPU-intensive activity. Instead of sampling the CPU at a specified rate, the interactive governor checks whether to scale the CPU frequency up soon after coming out of idle state by configuring a timer within 1-2 ticks.

The frequency set by the above power governors can be overwritten by the `thermal daemon` if platform temperature raises above the threshold. For example, on the Odroid-XU3, if the platform temperature goes above 95°C (default threshold), the operating frequency gets scaled down to 900 MHz until the platform temperature falls below the threshold.

2.3.2 RTM of Homogeneous Multi-cores

There are several works on offline optimization to achieve performance-energy trade-off points for homogeneous multi-cores by employing DVFS and/or task mapping [4, 41, 96, 97]. In the offline DVFS approach, pre-characterised workloads of a given application are used during runtime to adjust the power control levers at regular intervals for minimizing energy. In [41], design and implementation of a compiler-directed Dynamic Voltage Scaling (DVS) algorithm are presented for reducing the energy usage of memory-bound applications. The algorithm identifies program regions with significant memory stalls, where the CPU can be slowed down with a negligible performance penalty (0% to 5%). Choi *et al.* [4] presented a DVFS technique for MPEG decoding to reduce energy consumption while maintaining a Quality of Service (QoS) constraint. It predicts and characterizes the workload of an incoming frame by using a frame-based history to appropriately scale the processor voltage and frequency to provide the exact amount of computing power needed to decode the frame. A similar practical voltage scaling technique for mobile devices primarily running multimedia application is proposed in [96].

This approach extends the real-time scheduling algorithm by changing the CPU speed of task execution based on allocated and actual cycles of a task.

Considering that workload prediction techniques used for video processing applications cannot be applied to interactive games, Gu *et al.* [97] proposed a control theory-based DVS algorithm for interactive 3D game applications running on battery-powered portable devices. It predicts the workload of processing frame using Proportional-Integral-Derivative (PID) predictor and sets the voltage-frequency accordingly. In [67], an accurate and scalable method for determining the optimal system operating points (number of threads and DVFS settings) for parallel workloads is proposed. The operating point is computed, considering a set of objective functions (minimize delay or energy-delay product) and constraints (peak power or delay), to optimize for energy efficiency. This approach estimates the optimal system settings as a function of workload characteristics using Multinomial Logistic Regression (MLR) models. This also applies L1-regularization to to determine the relevant workload metrics for energy optimization automatically. Wildermann *et al.* [98] presented an analysis and a realistic model that relies on only limited local communication for negotiation of cores to find an optimal decentralized core allocation for many-core systems. This approach mainly focuses on maximizing the average speedup of running applications without considering other general metrics such as energy, performance constraints, etc.

Energy, power, and performance characteristics change dramatically as a function of workload. Moreover, a diverse set of workloads (e.g., scientific computing, financial applications, and media processing) makes the workload characterization using offline techniques inefficient. In such cases, online optimization techniques have been proved to be more effective [20, 35, 47, 99, 100]. Online optimization has been considered to cater for dynamic workload scenarios to optimize energy consumption while respecting the timing constraint. For online optimization, either all the processing is performed at runtime or else the optimization is supported by offline analysed results. Depending on the control mechanism, an online approach can be reactive or proactive. In the reactive approach, the *V-f* scaling is controlled based on the history of CPU workloads, where *V-f* is increased/decreased when CPU workload is higher/lower than a predefined value [9]. In the proactive approach, predicted workloads are used to manage the hardware power control levers [99, 101] with continuous adjustment through feedback from the hardware performance monitors.

Jejurikar *et al.* [99] proposed an energy efficient algorithm to compute task slowdown factors based on the contribution of processor leakage and standby energy consumption of the resources in a system. The presented algorithm has two phases: (1) computing the critical speed for each task; and (2) increasing the task slowdown factors if the task set is not feasible. Critical speed of a task is set by computing the energy consumption of each task at all possible discrete slowdown factors and choosing the factor that minimizes the

task energy. In [100], a technique for Continuous Frequency Adjustment (CFA) of various functional blocks in the system at a very low granularity to minimize energy while meeting a performance constraint, is proposed. The basic idea of CFA is to eliminate the power and delay costs incurred by the power-mode transitions which involve clock generators (e.g., phase-locked loop). The frequency is adjusted based on the predicted workload of a task, which is formulated as an Initial Value Problem (IVP) [102]. Choi *et al.* [103] presented a similar approach for task workload classification and *V-f* control with the feedback from performance counters. In this DVFS technique, the workload of a task is dynamically decomposed into on-chip and off-chip by using an embedded hardware unit in the processor (PMU). It also presents an accurate and efficient calculation of the ratio of on-chip computation time to off-chip access times by using information about the data cache misses and the CPU stall cycles due to data dependencies.

Weissel *et al.* [47] presented an energy-aware scheduling policy for non-real-time operating systems using PMCs. Based on the information extracted from the PMCs, the scheduler determines the appropriate frequency for each thread running in a time-sharing environment. A recurrent analysis of the thread-specific energy and performance profile allows an adjustment of the frequency to changes in application workload with a little loss in performance. The authors of [48] present an accurate runtime prediction of execution time and a corresponding DVFS technique based on memory resource utilization. This approach improves the accuracy of runtime predictions by using task execution time and cache miss rates as feedback. Cache miss rates indicate the frequency of memory accesses and enable us to derive the latencies introduced by these operations. A similar approach, which is a hardware-specific implementation of the stall-based model, is proposed in [49]. It uses the processor performance model to scale the frequency, expressed as a change (in cycles) of the main memory latency.

Siyu *et al.* [104] and Shen *et al.* [105] have proposed online approaches using machine learning algorithm for selecting optimal *V-f* level required for an application in the presence of performance variations due to application-generated CPU workloads. However, aforementioned approaches cannot adapt to the variations in application performances, such as frame rate for video decoders, page loading rate for browsers, etc. Moreover, existing approaches using machine learning [104, 105] use a single runtime formulation of VFS for a given performance requirement, which cannot cater to intra- and inter-application workload variations. Therefore, to adapt to such changes, energy minimization approaches [3, 106] using Reinforcement Learning (RL) algorithm to suitably control *V-f* for a given performance requirement are proposed. Furthermore, to reduce the relearning time and Q-table size for new workloads/applications, learning transfer is proposed by extending the RL algorithm [20]. These approaches perform well for unknown applications to be executed at runtime but lead to inefficient results as optimization decisions need to be taken quickly and offline analysis results are not used.

Further, they are agnostic of concurrent workload variations and thus fail to adapt for concurrently executing multiple applications.

There has also been a focus on online optimization facilitated by offline analysis results [5, 40, 107, 108]. Such approaches lead to better performance results than only online optimizations as they take advantage from both offline and online computations. In [40], thread-to-core mapping, and DVFS are performed to maximize performance within a power budget. To capture the workload dependence of the performance-power Pareto frontier, an MLR classifier is built offline using performance counter, temperature, and power characterization data. During runtime, based on sampled PMC and temperature data, the classifier selects the optimal operating point that maximizes the performance within a power budget. Sasaki *et al.* [5] presented a similar approach for runtime coordinated power-performance management system called C-3PO, which optimizes the performance of many-core processors under a power constraint by controlling two software knobs: thread packing and DVFS. This approach distributes the power budget to each program by controlling the threads to be executed with an appropriate number of cores and operating frequency. The power budget is distributed carefully in the form of the number of allocated cores or operating frequency depending on the power-performance characteristics of the workload so that each program can effectively convert the power into performance. However, the aforementioned two approaches do not consider the concurrent workload variations in setting *V-f* level and are not suitable for cluster-based DVFS multi-cores (where a group of cores is set to the same *V-f* level).

Quan and Pimentel [108] proposed a scenario-based runtime mapping approach targeting homogeneous MPSoCs in which mappings derived from design-time DSE are stored for runtime mapping decisions. It uses clustering method for grouping the similar inter-application scenarios and only storing a single mapping for each cluster of workload that results in best performance. Additionally, this approach does online mapping optimization by continuously monitoring the system and doing mapping customizations to improve the system performance further.

Techniques proposed in [109–111] determines bottlenecks during application execution using performance counters. They present a framework for direct, automatic profiling of power consumption for non-interactive and parallel scientific applications to assist the scheduler. Rountree *et al.* [112] do a critical path analysis to determine which tasks may be slowed down to achieve energy savings while minimizing the performance loss in the parallel execution. This analysis appears beneficial only when applications have computation or communication imbalances among participating processes, which is typically not the case for highly efficient parallel applications [113].

The schemes presented in [114,115] determine the communication phases to apply DVFS. A technique that applies both DVFS and over-clocking to CPUs to save energy and improve execution time is discussed in [116]. Marathe *et al.* [117] proposed a runtime system

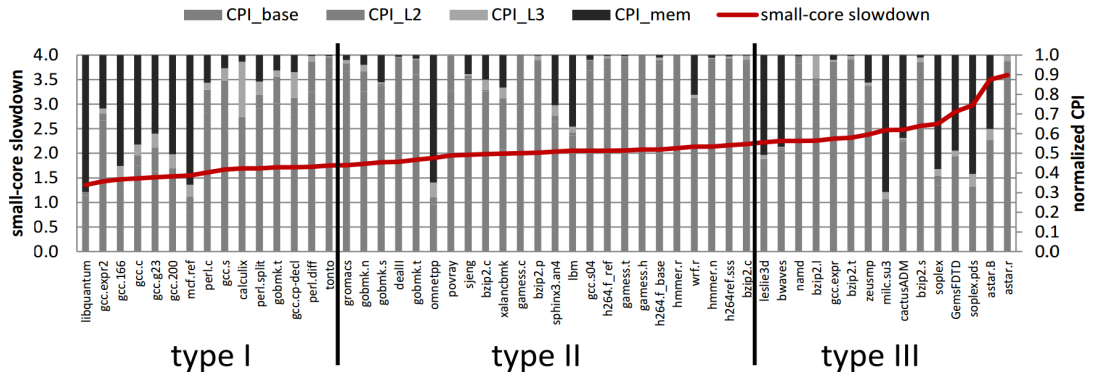

FIGURE 2.11: Normalized big-core CPI stacks (right axis) and small-core slowdown (left axis). Benchmarks are sorted by their small-versus-big core slowdown. Reprinted from [10].

conductor that dynamically distributes available power to different compute nodes and cores based on the available slack to improve performance. The conductor performs both upscaling and downscaling of processor frequency to decrease execution time and to save energy in an indirect manner through power clamping, which differs from the traditional approach of only downscaling to save energy. The authors of [118] proposed a latency-aware DVFS algorithm to avoid aggressive power state transitions. They argue that too frequent DVFS changes are not only unprofitable but also detrimental to performance, due to the extra time and energy costs introduced. This approach divides each application into phases through profiling and uses this information to decide whether changing the *V-f* setting is beneficial or not during the application execution.

Sundriyal *et al.* [119] present an energy management approach, relying on the Intel Running Average Power Limit (RAPL). It provides a standard interface for measuring and limiting the processor and memory power. This approach uses Memory Access Per Micro-operation (MAPM) and MIPS (millions of instructions per second) metrics to indicate how a change of frequency affects the performance. A procedure to select power capping thresholds dynamically on the Xeon Phi platform is given in [113]. They noticed that default power capping limits employed on this platform are much higher than the majority of applications would reach. Considering this, different power limits are defined according to the workload characteristics and application performance.

2.3.3 RTM of Heterogeneous Multi-cores

Heterogeneous multi-cores can enable higher performance and reduced energy consumption by executing workloads on the most appropriate core type. Fig. 2.11 compares the slowdown for SPEC CPU2006 [120] workloads on a small core relative to a big core (left y-axis), to the normalized Cycles Per Instruction (CPI) stack [121] on a big core (right y-axis). The normalized CPI stack indicates whether a workload is memory-intensive or compute-intensive. If the normalized CPI stack is memory dominant, then the workload

is memory-intensive (e.g., mcf), else the workload is compute-intensive (e.g., astar). The figure illustrates workloads grouped into three categories on the x-axis: workloads that have reasonable slowdown (<1.75×) on the small core (type-I workloads), workloads that have significant slowdown (>2.25×) on the small core (type-III), and the remaining workloads are labeled as type-II. Making correct scheduling decisions in the presence of type-I and III workloads is most critical: making an incorrect scheduling decision, i.e., executing a type-III workload on a small core instead of a type-I workload, leads to poor overall performance. Therefore, mapping a workload onto an appropriate core type is essential to improve performance and energy efficiency in heterogeneous multi-cores.

Goraczko *et al.* [43] presented a resource model to improve the accuracy of existing models considering the time and energy costs of runtime mode switching. For a given application, the software partitioning problem (assign parts of an application to each processor to achieve maximum system lifetime without sacrificing application performance) has been formulated as an Integer Linear Programming (ILP) problem. The approach presented in [107] generates multiple mappings for each application with a trade-off between resource requirement and throughput. It contains an offline, which computes template mappings for each scenario sequence by applying the multi-step approach developed for task assignment, independently from the actual scenario. At runtime, a manager observes mode changes and chooses an appropriate precomputed template mapping. The evaluation considers only power consumption and does not report any performance values. In a similar direction, in [122], a domain-specific hybrid task mapping (HTM) algorithm for heterogeneous multimedia MPSoCs that relies heavily on offline analysis result to capture dynamic behavior both between and within applications. Above two approaches have been validated through simulation.

In [123], an adaptive energy minimization approach for heterogeneous embedded systems is presented, through regression-based learning of energy/performance tradeoffs between different computing resources in the system. Using this model, an application task is mapped onto a computing resource that minimizes energy consumption while meeting application performance requirement. It also performs DVFS to adapt to performance and workload variations.

Craeynest *et al.* [10] proposed a Performance Impact Estimation (PIE) technique to map the workload onto an appropriate core type in a heterogeneous multi-core processor. The mapping is done by estimating the expected performance for each core type for a given workload. PIE collects CPI stack, Memory-Level Parallelism (MLP) and Instruction-Level Parallelism (ILP) profile information during runtime on any one core type, and estimates performance if the workload were to run on another core type (impact on overall performance). Dynamic PIE scheduling collects profile information on a per-interval basis and dynamically adjusts the workload-to-core mapping, thereby exploiting time-varying execution behavior. Ma *et al.* [51] proposed a comprehensive analytical model to predict the power efficiency from thread-level parallelism (scale-out)

and enabling thread migration between heterogeneous cores (scale-up). The model is composed of a performance and a power model. The performance model is built by two orthogonal functions, consisting of scale-out speedup from multi-threading and scale-up speedup from core heterogeneity. The power model estimates the power consumption of corresponding scale-out and scale-up configurations. It simultaneously captures the power variations caused by thread synchronization and thread migration between heterogeneous cores.

Aalsaud *et al.* [50] presented a runtime optimization approach to maximize power normalized performance considering dynamic variation of workload and application scenarios. It involves extensive offline analysis to identify the tradeoffs between inter-application concurrency with performance and power consumption under different thread-to-core mappings and frequencies. Then, using this experimental data collected on an Odroid-XU3 heterogeneous platform with a number of PARSEC benchmark applications, a model is presented for power normalized performance, i.e., Instructions per Second (IPS)/Watt), underpinning analytical power and performance models, derived through multivariate linear regression.

In a similar direction, Gupta *et al.* [124] presented an approach, called DyPO, to find the Pareto-optimal configurations at runtime as a function of the workload. This approach involves application instrumentation to identify workload snippets that from different workload phases; offline characterisation data collection to find classifiers that map each workload snippet to an optimal configuration; runtime selection to feed the classifiers with runtime statistics to find the optimal configuration. In [52], Sozzo *et al.* proposed a new runtime resource management policy for ARM's big.LITTLE architecture running a single parallel application at a time. This policy exploits DVFS and dynamic thread management, to minimize power consumption under performance constraints by balancing the thread execution and adapting to workload variations.

Donyanavard *et al.* [19] presented SPARTA, a throughput aware runtime task allocation approach for heterogeneous many-core platforms to achieve energy efficiency. It uses a classification-based prediction method for predicting tasks' behavior across different core types that model runtime performance/power (instructions-per-Joule). Based on the predicted performance/power, it then maps the tasks onto the cores that maximize the energy efficiency. Becchi *et al.* [125] presented a dynamic policy to map running tasks to processor cores to maximize the performance by accurately exploiting available heterogeneity of the system. To equally share the resources, the policy observes runtime behavior of the running threads and migrates the threads between the cores in a round-robin fashion or based on estimated Instructions Per Cycle (IPC).

Chen *et al.* [126] presented a framework to leverage the fundamental relationship between the inherent program characteristics and the corresponding resource demands for scheduling an application. This addresses three aspects of a program scheduler in

TABLE 2.1: Comparison of various run-time management approaches for homogeneous (homo.) and heterogeneous (heter.) multi-core architectures

Reference	Architecture	Mapping	Threads Distribution	Adaptive DVFS	Approach Type	Application Execution Scenario	Concurrent Workload Classification	Adaptation at Arrival/completion times	Evaluation Application Suite
[4, 41, 96, 97]	Homo.	No	N/A	Yes	Offline	Single	N/A	No	MPEG, SPEC CFP95, MPEG, MPGDec, H263Enc, H263Dec
[98]	Homo.	Yes	N/A	No	Offline	Concurrent	No	No	Not given
[115]	Homo.	No	N/A	Yes	Offline	Single	N/A	No	NAS Parallel
[99, 100]	Homo.	No	N/A	Yes	Online	Single	N/A	No	Randomly generated tasks, Network controller
[3, 20, 104–106]	Homo.	No	N/A	Yes	Online	Single	N/A	No	ffmpeg, WLAN module, MiBench, Web-browser, Mplayer
[35]	Homo.	Yes	N/A	Yes	Online	Single	N/A	No	PARSEC, MiBench, SPLASH-2
[47–49]	Homo.	No	N/A	Yes	Hybrid	Single	N/A	No	'find|grep', gzip, djpeg, cjpeg, factor, SPEC CPU2006
[67]	Homo.	Yes	N/A	Yes	Hybrid	Single	N/A	No	PARSEC
[40]	Homo.	Yes	N/A	Yes	Hybrid	Single/Concurrent	No	No	PARSEC
[108]	Homo.	Yes	N/A	No	Hybrid	Single	N/A	No	MJPEG, Sobel, MP3
[5]	Homo.	Yes	N/A	Yes	Hybrid	Single/ Concurrent	No	No	PARSEC
[109, 110]	Homo.	No	N/A	Yes	Hybrid	Single	N/A	No	NAS Parallel, SPEC CPU2000, SPEC CFP95
[114, 119]	Homo.	No	N/A	Yes	Hybrid	Single	N/A	No	NAS Parallel (MPI Programs only)
[43]	Heter.	Yes	One core type	No	Offline	Single	N/A	No	Sound Source Localization (SSL) App.
[107, 122]	Heter.	Yes	One core type	No	Hybrid	Single/ Concurrent	No	No	WLAN, DVB-H, UWB, MJPEG encoder, MP3 decoder, Sobel filter
[123]	Heter.	Yes	One core type	Yes	Online	Single	N/A	No	Image processing
[125–127]	Heter.	Yes	One core type	No	Hybrid	Single/ Concurrent	No	No	SPEC2000
[10]	Heter.	Yes	One core type	No	Hybrid	Single/ Concurrent	Yes	No	SPEC CPU2006
[51]	Heter.	Yes	One core type	No	Hybrid	Single/ Concurrent	No	No	PARSEC
[19]	Heter.	Yes	One core type	Yes	Hybrid	Single/ Concurrent	Yes	No	MiBench, PARSEC
[50]	Heter.	Yes	Two core types	No	Hybrid	Single/ Concurrent	Yes	No	PARSEC
[52]	Heter.	Yes	One core type	Yes	Hybrid	Single	N/A	No	Blackscholes
[124]	Heter.	Yes	Two core types	Yes	Hybrid	Single/ Concurrent	Yes	No	MiBench, PARSEC
This work	Homo./Heter.	Yes	Two core types	Yes	hybrid	Concurrent	Yes	Yes	MiBench, ALPBench, SPEC CPU2006, PARSEC, SPLASH-2, RoyLongbottom, LMBench, Rodinia, and NAS Parallel

the heterogeneous multi-processing context: understanding the physical configurations of the core supply, estimating the resource demands of the running applications, and identifying the program-to-core matching for a given criterion. The method identifies program-to-core matching with Weighted Euclidean Distances (WED), by projecting core configurations and the program's resource demands into a unified multi-dimensional space. In [127], Koufaty *et al.* proposed a bias scheduling for heterogeneous systems with cores that have different microarchitectures and performance. By dynamically monitoring application bias (difference observed when executing on cores of a different type), the operating system matches threads to the core type that can maximize system throughput. Bias scheduling influences the existing scheduler to select the core type that best suits the application when performing load balancing operations. Moreover, this technique does not require sampling of CPI on all core types or offline profiling.

To present the features of recently reported approaches that are closely related to the investigations presented in this thesis, Table 2.1 shows a summary and comparison of these approaches that consider run-time management for multi-core architectures. Column I gives the references corresponding to the approaches, and Column II presents the architecture type considered (Homogeneous (Homo.) or Heterogeneous (Heter.), while Column III and IV show mapping and thread distribution strategy for multi-threaded applications – mapping entirely on one core type or simultaneously allocating multiple types of cores, respectively. Column V gives whether an approach adapts to workload variations by periodically adjusting the DVFS settings or not. Kind of technique (offline, online, or hybrid (online + offline)) is presented in Column VI. Details about application execution scenario (single – executing one application at a time, or concurrent – running multiple applications simultaneously) are shown in Column VII. Information about workload classification of concurrently executing applications is given in Column VIII, while Column IX shows whether an approach adapts to application arrival or completion at runtime in a concurrent execution scenario. Finally, Column X presents applications used for evaluating an approach.

From the table, approaches targeting homogeneous architectures had mostly considered single-application scenarios [3, 4, 20, 35, 41, 47–49, 67, 96–100, 104–106, 115]. The works that consider the concurrent execution scenario have the following drawbacks. For example, [98] concentrates only on thread-to-core mapping to maximize average speedup, and do not consider workload variations to adjust the DVFS settings and other important metrics such as performance constraints and energy consumption. Whereas, approaches [40] and [5] try to meet power budget while maximizing the performance by increasing/decreasing frequencies and cores. These approaches also do not adapt to concurrent workload variations along with application interference. As a result, such approaches misses the opportunity to minimize energy consumption and to improve performance.

The aforementioned works that consider multi-threaded applications, executing on heterogeneous multi-cores, map threads of an application completely onto one type of core situated within a cluster [10] [51] [52] [19]. This reduces the thread-to-core mapping complexity, but misses to benefit from the distribution of an application threads to multiple types of cores at a time. In a similar direction, approaches [125–127] used workload memory intensity as an indicator to guide thread-to-core mapping. For example, a given platform containing two types of cores as big and LITTLE, such proposals map memory-intensive workloads on a LITTLE core and compute-intensive workloads on a big core. Unlike [10, 51], techniques proposed in [19, 50, 52] exploit DVFS, but they have several drawbacks. For example, in [52], design space is explored for a single application, which increases exponentially if concurrent applications have to be considered. In [19], each application is executed as single threaded and use only one type of core for it at a time.

In [50, 124], threads of an application are mapped onto more than one type of cores, but the approach heavily depends on offline regression analysis of performance and energy for all possible thread-to-core mappings and *V-f* settings, which is non-scalable. Additionally, in [50], *V-f* setting is not adjusted during execution, which is beneficial for adapting to workload variations. Approach presented in [124] needs extensive analysis and instrumentation of each application into basic blocks for collecting the PMC data during runtime and does not consider application performance constraints. Above two approaches do not perform adaptive mapping at application arrival/exit, and thus it is not efficient if a new and unknown application arrives or existing application finishes. The following chapters focus on addressing the above problems related to both homogeneous and heterogeneous multi-cores.

The table also shows applications used for the evaluation in different works. To facilitate the comparison with the existing works, the following benchmarks that are highly used for the experimental validation or for training the models are selected: MiBench, SPEC CPU2006, PARSEC, ALPBench, SPLASH-2, RoyLongbottom, LMBench, Rodinia, NAS Parallel. The details of the benchmarks are given in Section 2.4.2.

2.4 Experimental Platforms, Applications and Tools

This section presents the details of hardware platforms and applications used to conduct experiments in the following chapters. Further, tools that are used for analysing the characterisation data and/or implementing the proposed approaches are detailed.

2.4.1 Experimental Platforms

For experimental analysis and validation of the proposed approaches, three different platforms have been considered. They have different characteristics in terms of ISA,

microarchitecture, platform DVFS type, number of processing cores. For example, the Odroid-XU3 [28] has a state-of-the-art mobile SoC with the popular ARM big.LITTLE heterogeneous processing architecture, which is prevalent in tablet/smartphone devices). In contrast, the Intel Xeon E5-2630V2 and Xeon Phi 7120P are homogeneous multi-core platforms. All platforms offer a wide variety of platform configurations such as core enabling/disabling, multiple DVFS settings per cluster and support real-time sampling of power sensors and hardware PMCs. Furthermore, different types of DVFS architecture are supported, where the Odroid-XU3 has cluster-wide DVFS, Xeon E5-2630V2 supports per-core DVFS, and Xeon Phi 7120P is based on system-wide DVFS. In the case of cluster-/system-wide DVFS platform, the whole cluster or system with a group of cores is set to operate at a single DVFS level. Whereas, for per-core DVFS platforms, individual cores can run at different DVFS settings at a time. Further details of each platform are given below.

2.4.1.1 Odroid-XU3

The modern heterogeneous architectures contain different types of cores in varying number. One such architecture is considered for this work, which is a 28nm Samsung Exynos 5422 System-on-Chip (SoC) [128] hosted on the Odroid XU3 board [28]. It is based on the ARM's big.LITTLE heterogeneous architecture and contains two clusters named big and LITTLE [129]. Each cluster offers different tradeoffs between power and performance. Figure 2.12 shows the Odroid-XU3 board emphasizing the Exynos 5422 SoC. In addition, the chip contains a Mali-T628 GPU and 2 GB DRAM LPDDR3. The big and LITTLE clusters contain high performance Cortex-A15 quad-core processor and low power Cortex-A7 quad-core processor, respectively. The four Cortex-A15 cores each have 32 KB instruction and data caches and share a 2 MB L2-cache. The four Cortex-A7 cores each have 32 KB instruction and data caches, and share a 512 KB L2-cache.

The board also contains four real-time current/voltage sensors that facilitate measurement of power consumption (static and dynamic) on the four separate power domains: big (A15) cores, LITTLE (A7) cores, GPU and DRAM. The Odroid-XU3 board can run different flavors of Linux. It also supports core disabling and DVFS, helping in optimizing system operation in terms of performance and energy consumption. DVFS can be used to change *V-f* levels at a per-cluster granularity. For each power domain available for a cluster, the supply voltage and clock frequency can be adjusted to pre-set pairs of values. The Cortex-A15cluster has a range of frequencies between 200 MHz and 2000 MHz with a 100 MHz step, whereas the Cortex-A7 cluster can adjust its frequencies between 200 MHz and 1400 MHz with a step of 100 MHz. The device firmware automatically adjusts the voltage for a selected frequency. The availability of multiple heterogeneous processing elements and accurate power measurement sensors makes this an ideal platform for validating the runtime energy management techniques.

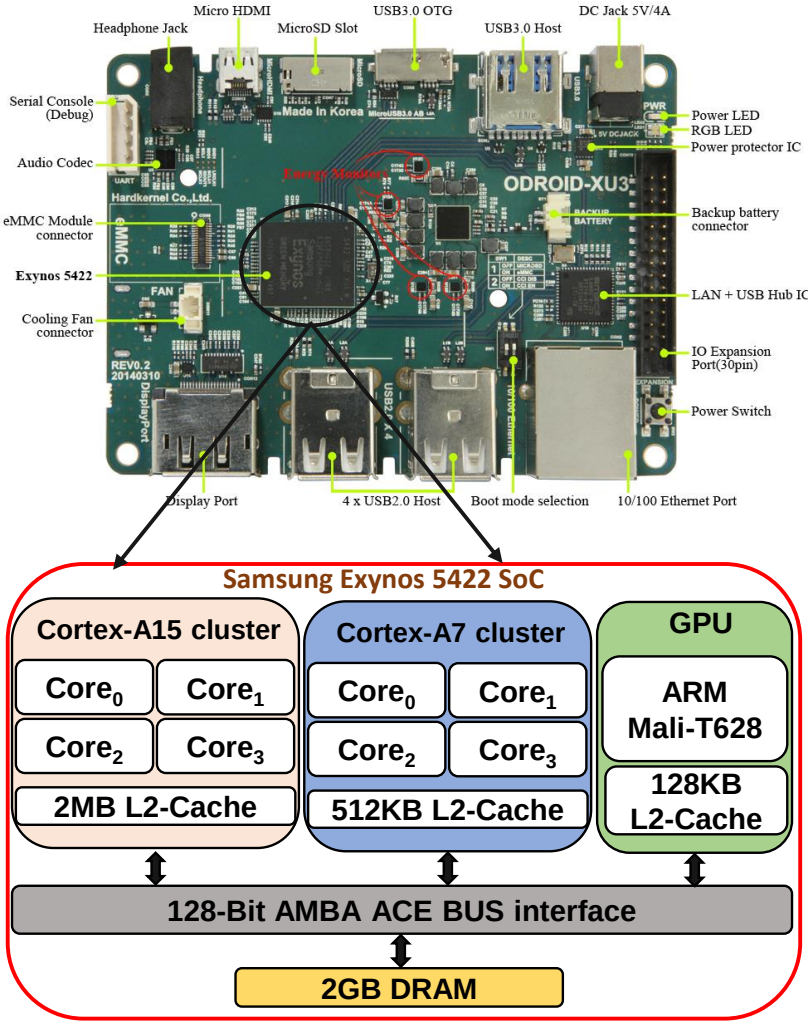

FIGURE 2.12: Odroid-XU3 board (top) containing Samsung Exynos 5422 heterogeneous MPSoC (bottom). Adapted from [11].

2.4.1.2 Intel Xeon E5-2630V2

The Intel Xeon E5-2630V2 is a 64-bit multi-core enterprise processor built on 22 nm process technology. This processor is based on high performance Ivy Bridge (IVB) microarchitecture. The block diagram of the two-socket Xeon E5-2630V2 processor is shown in Fig. 2.13. It has a bi-directional ring bus that connects all of the components (cores, L3, memory controller, QuickPath Interconnec (QPI) and Peripheral Component Interconnect express (PCIe) controller). Each socket contains 6-cores and platform has a total of 12 physical cores or 24 logical cores with hyper-threading enabled. It has three levels of cache hierarchy with 32 KB of L1 (I/D), 256 KB of L2 and 15 MB of L3, and 32 GB of main memory running at 1600 MHz. The processor supports per-core DVFS with 15 levels ranging from 1.2 GHz to 2.6 GHz in 100 MHz steps.

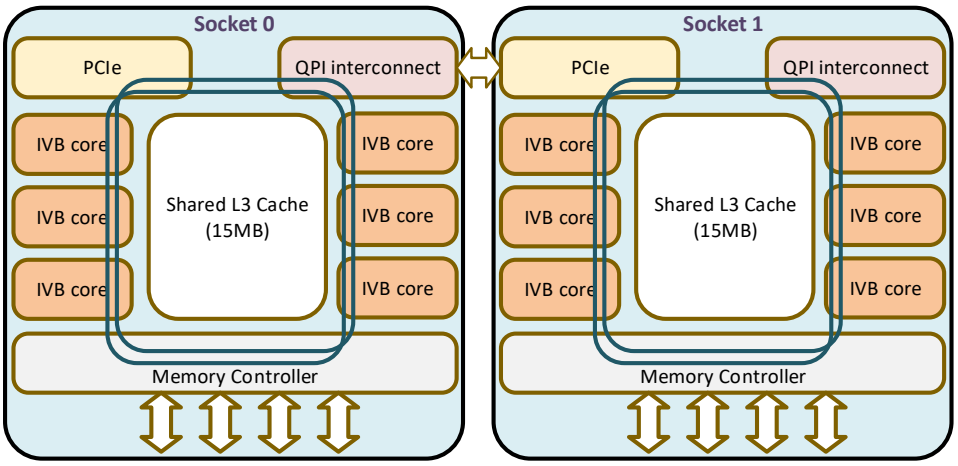

FIGURE 2.13: The block diagram of the Intel Xeon E5-2630V2. Redrawn from [12].

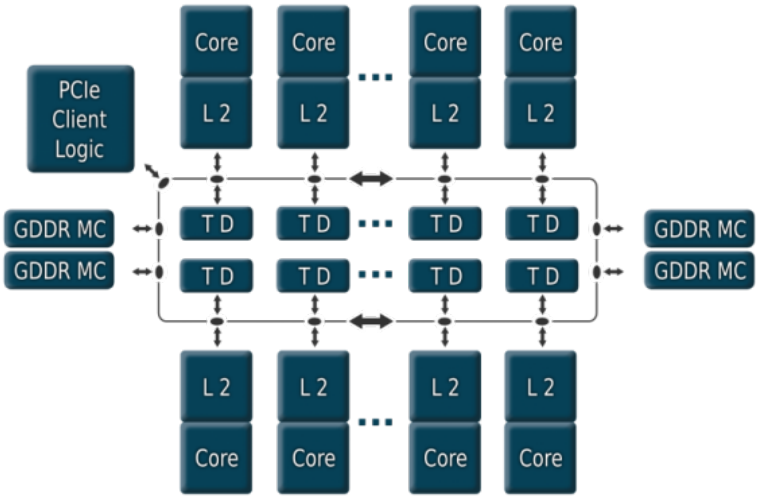

FIGURE 2.14: The Intel Xeon Phi chip architecture layout with ring-based interconnect. Taken from [13].

2.4.1.3 Intel Xeon Phi 7120P

Intel Xeon Phi 7120P is based on Intel's Many Integrated Core (MIC) architecture, containing 61 cores connected by a high performance on-die bidirectional interconnect. The coprocessor is connected to an Intel Xeon processor – the "host" – via the PICe bus. Fig. 2.14 depicts the architecture layout of the coprocessor. The processor core is an in-order architecture (scalar unit), which is based on the Intel Pentium processor family. Each core can fetch and decode instructions from four hardware threads, i.e., each core supports four-way hyper-threading. There are eight memory controllers supporting up to 16 GDDR5 channels, and a theoretical aggregated bandwidth of 352 GB/s is provided. The L1 cache has a 32 KB L1 instruction cache and 32 KB L1 data cache. Associativity is 8-way, with a cache line-size of 64 bytes. The L2 cache is a unified cache which is

inclusive of the L1 data and instruction caches. Each core contributes 512 KB of L2 to the total global shared L2 cache storage. If no cores share any data or code, then the effective total L2 size of the chip is up to 31 MB. On the other hand, if every core shares the same code and data in perfect synchronization, then the effective total L2 size of the chip is only 512 KB. The actual size of the workload-perceived L2 storage is a function of the degree of code and data sharing among cores and thread. Like for the L1 cache, associativity is 8-way, with a cache line-size of 64 bytes.

Details about the system startup and the network configuration can be found in [130] and in the documentation coming with the Intel Manycore Platform Software Stack (Intel MPSS) [131]. Fig. 2.14 also shows the interface to the GDDR5 main memory of 16 GB and the global-distributed tag directories (TD) which ensure cache coherency. All cores share a common Voltage-frequency island, and the frequency can be varied from 619 MHz to 1238 MHz in nine steps with corresponding voltage ranging from 0.995 V to 1.060 V. To measure the energy consumption of the cores and main memory, the coprocessor has the read-only model specific register (MSR) `MSR_PP0_ENERGY_STATUS` and `MSR_DRAM_ENERGY_STATUS`. These MSRs are updated every 1 ms with a wraparound time of around 60 secs when power consumption is high and maybe longer otherwise.

2.4.2 Applications

To understand the performance of the underlying hardware platform in terms of execution time and power to different workloads, selection of a diverse set of application is essential. Therefore, a set of benchmarks have been considered in the following chapters, and their details are given in this section.

MiBench [132] consists of six suits (35 single-threaded applications) with each representing a specific area of the embedded domain. The six categories are automotive and industrial control, consumer devices, office automation, networking, security, and telecommunications. Each suite has a varying number of applications, and small and large input data set representing embedded and real-world applications respectively.

ALPBench [133] is composed of five complex multi-threaded multimedia applications, namely speech recognition (CMU Sphinx 3), face recognition (CSU), ray tracing (Tachyon), MPEG-2 encoder and MPEG-2 decoder (MSSG). It uses traditional POSIX threads (pthreads) based implementation and supports changing the thread count for each application at compile time.

SPEC CPU2006 [120] has a wide range of industry standardized applications with diverse workload profiles [49, 134]. This benchmark suite is industry-standardized, stressing a system's processor, memory subsystem and compiler. SPEC designed this suite to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications. It has

twelve *integer benchmarks*: perlbench, bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, h264ref, omnetpp, astar, xalancbmk; eighteen *floating point benchmarks*: bwaves, gamess, milc, zeusmp, gromacs, cactusADM, leslie3d, namd, dealII, soplex, povray, calculix, GemsFDTD, tonto, lbm, wrf, sphinx3, specrand.

PARSEC [15] is a conventional RMS (Recognition, Mining, and Synthesis) benchmark suite and representatives of emerging large-scale multi-threaded commercial programs, as envisioned by the Intel a decade ago. It is a collection of 31 applications from diverse areas such as video encoding, computer vision, financial analytics, image processing, and animation physics. All the applications are parallelised, supporting multi-threading for MPSoC platforms and configuring the number of threads through a command-line argument. It is targeted for emerging workloads, which have already become essential applications over the years. Moreover, it focuses on applications from all domains, such as server and desktop applications. This suite is primarily intended for research, and it can easily be used for performance measurements.

SPLASH-2 [18] is a suite of eleven parallel applications suite that has been released to facilitate the study of centralized and distributed shared-address-space multiprocessors. The suite is composed of multi-threaded applications from high-performance scientific computing, signal processing and graphics domains, which are based on the original SPLASH-2 benchmarks suite (1995) and includes changes to make this suite compatible with modern programming practices to create and handle parallelism. SPLASH-2 has been the most commonly used suite for research in parallel programming with shared-memory systems.

RoyLongbottom [135] benchmark collection is a free set of programs that measure performance and reliability of various components of a computing system, such as CPUs, caches, memory, disks, and graphics.

LMBench [136] is a suite of simple, portable, ANSI/C microbenchmarks for UNIX/-POSIX. In general, it measures two key features: latency and bandwidth. lmbench is intended to give system developers insight into basic costs of key operations.

Rodinia [16] benchmark suite is developed for heterogeneous architectures and accelerators by the University of Virginia. The benchmarks are inspired by the Berkeley dwarfs taxonomy, featuring application kernels that have been parallelised for multi-core CPU and GPU platforms. The suite has diverse workload profiles, encompassing a range of parallel communication patterns, synchronisation techniques, and power consumption patterns.

NAS Parallel Benchmarks (NPB) [17] are a small set of applications designed to help evaluate the high performance computing systems and supports commonly-used programming models like MPI and OpenMP. The first version has five kernels and three pseudo-applications, derived from computational fluid dynamics (CFD) applications.

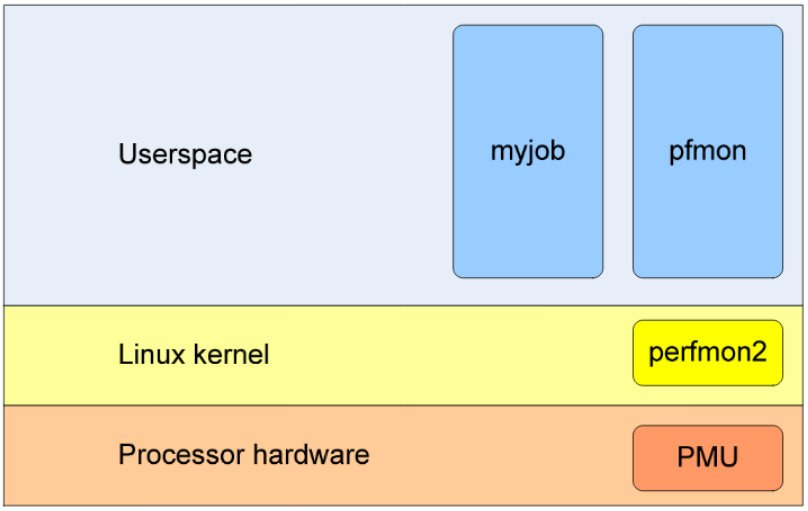

FIGURE 2.15: `Perfmon2` and `pfmon`, sitting in Kernel and Userpsace level, respectively. Reprinted from [14].

The suite has been extended to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids.

The benchmarks above provide applications from different domains (e.g., embedded, multimedia, high-performance computing, etc.), implementation (single-/multi-threaded), and varying synchronisation overheads with distinct data partitions and sharing patterns. Moreover, benchmarks such as RoyLongbottom and LMBench are more suitable for platform characterisation by stressing various parts of the CPU. MiBench and ALPbench cover applications from embedded and multimedia, while PARSEC and SPLASH represent emerging applications with multi-threaded implementation and relatively large execution times. Rodinia and NPB are mostly used for high-performance computing research and suitable for running on platforms with more cores (e.g., Intel Xeon Phi).

2.4.3 Tools for Monitoring and Analysis of Performance Events

To make a runtime decisions efficiently, the application characteristics, such as cache misses, CPU cycles, have to be monitored during the application execution or at design time. This necessitates a standard interface for accessing the Performance Monitoring Unit (PMU) on the processor. Therefore, the `perfmon2` tool [14] is used for monitoring various performance events during design time and runtime. Furthermore, to analyse the collected performance events to understand their relationship with the performance and energy, and to build models, WEKA workbench [137].

Perfmon2 tool [14] is a software component that is about to be added to the Linux kernel as the standard interface to the PMU on popular processors, including ARM, x86 (AMD and Intel), Sun SPARC, MIPS, IBM Power, and Intel Itanium. It is designed to

give full access to a given PMU and all the corresponding hardware performance counters. Typically the PMU hardware implementations use a different number of registers, counters with different length and possibly other unique features. Although processors have different PMU implementations, they typically use configurations registers and data registers. `Perfmon2` provides a uniform abstract model of these registers and exports read/write operations accordingly. `libpfm` library and `pfmon` complement `perfmon2` by enabling the user applications to access (read/write) the `perfmon2` exported registers. Fig. 2.15 shows `Perfmon2` and `pfmon` as part of kernel space and userspace, respectively. This tool will be used to measure the metrics that are essential to classify the benchmarks. As experimentally shown in Section 3.1, various metrics contribute to variations in application workload. However, the metrics that have significant influence have to be monitored as hardware platforms support a limited set of counters (only seven, including the cycle counter, on the Odroid-XU3 and four on the Intel Xeon E5-2630V2). The important metrics are branch misses, L1-D/I cache misses, L2 cache misses, instructions retired, active CPU cycles, per core CPU utilisation, memory reads per instruction, number of active cores, the frequency of the cores and, in case NUMA-based platform, remote memory access.

WEKA Workbench [137] is a collection of machine learning algorithms and data preprocessing tools that include a set of algorithms described in [138]. WEKA was developed at the University of Waikato in New Zealand; the name stands for the Waikato Environment for Knowledge Analysis. It is designed so that you can quickly try out existing methods on new datasets in flexible ways. It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the result of learning. As well as a wide variety of learning algorithms, it includes a wide range of preprocessing tools. This diverse and comprehensive toolkit is accessed through a common interface so that its users can compare different methods and identify those that are most appropriate for the problem at hand. This tool is used for exploring various methods for building performance models and selecting the one with low overhead and better accuracy.

2.5 Discussion

The increased demand for power-efficiency and performance has led to powerful MPSoC platforms with a higher number of processing cores and heterogeneity across the cores. These platforms provide a range of operating points, such as DVFS levels and active core configurations that can be tuned at runtime to offer various power-performance trade-offs. These configurations have to be managed appropriately to be able to meet application performance constraints and/or energy efficiency. These platforms often execute multiple applications, with different performance requirements, concurrently.

These applications usually contend for the hardware resources (e.g., processing cores and memory bandwidth) to meet their performance requirements. Unlike in single application scenario, the concurrent execution of applications generates varying and mixed workloads due to shared resources. Therefore, the runtime management system plays a vital role in energy management under performance and resource constraints.

This chapter has presented a necessary background to appreciate the current trends and the contributions given in the following chapters. This includes covering low-power techniques, different MPSoC architectures, current trends in runtime energy management (summarized in Table 2.1), and finally hardware platforms, benchmark applications, and tools used for experimental analysis and validation.

State-of-the-art approaches to runtime energy management, considering homogeneous and/or heterogeneous MPSoC architectures, have been critically analysed to identify the areas where contributions can be made. These approaches usually control DVFS and/or thread-to-core mapping through offline, online, or hybrid methods to achieve energy efficiency. The offline approaches typically use computationally intensive design space exploration (DSE) methods to find an optimal or near-optimal configuration. Conversely, online/dynamic methods must not be computationally expensive, therefore uses heuristics to find a good solution. However, for obvious reasons, offline methods perform better than online approaches in finding DVFS settings and mappings. To address the shortcomings of pure offline and online methods, hybrid approaches have been widely used. The remainder of the thesis investigates various hybrid approaches that can cope with dynamic workloads, application execution scenarios (single, concurrent, and dynamic addition/exit of applications), and different hardware architectures.

As shown in Table 2.1, the existing approaches for homogeneous and heterogeneous MPSoCs do not efficiently exploit the DVFS potential of the cores, as they are unaware of concurrent workload variations (Research Question 1). Moreover, these approaches are not adaptable to diverse DVFS techniques (e.g., cluster-wide, per-core), dynamic addition and exit of applications. In case of heterogeneous multi-cores, the benefit of heterogeneity is not efficiently utilised, for example, simultaneously mapping an application onto more than one type of core(s), which may reduce the energy consumption (Research Question 2). This motivates to look into developing a runtime management system to manage concurrent execution scenarios energy-effectively under performance and workload variations (Chapter 4). Furthermore, as new applications have continuously been by application developers and smartphone users may install/uninstall different applications, it necessitates an application offline profiling-independent runtime management approach (Research Question 3). This has also been investigated in the Chapter 5. In the following chapters, the above issues related to homogeneous multi-core platform are addressed first, then the focus is shifted to heterogeneous multi-core platforms.

Chapter 3

Runtime Management of DVFS for Homogeneous Multi-cores

Chapter 2 has outlined that modern embedded systems need to deal with multiple applications concurrently and the existing approaches are not efficient in handling such scenarios, which is experimentally demonstrated in the following sections. Due to limited resources availability, when multiple applications are run at a time, they compete for resources (e.g., processing cores, memory bandwidth, etc.). Moreover, each application may have different performance requirements and exhibit workload variations during runtime. Therefore, the runtime energy management system should proactively select the Voltage-frequency (*V-f*) setting for adapting to the varying workload. To accomplish that, this thesis postulates that the runtime manager should be aware of the relation between *V-f*, workload, and performance requirements. Such a runtime manager requires efficient way of identifying the kind of workload present on the processor and mapping of that workload to an appropriate DVFS setting at runtime. As part of this, firstly, classification of application workload based on execution time and Performance Monitoring Counter (PMC) data for various application execution scenarios, is presented in Section 3.1. Then, based on this workload classification, an online energy management technique for concurrent execution of applications (single-threaded (Section 3.2) and multi-threaded (Section 3.3)) is discussed. To evaluate the approaches on different platforms and determine how they perform, initially homogeneous multi-core platforms with various execution scenarios are considered for the experimental analysis and validation.

3.1 Workload Classification

The application workload on the processor varies depending upon the dynamic instruction mix of an application (percentage of branches, loads, stores, etc.) [134]. As the

application executes, the proportion of various instructions may vary, thereby the underlying hardware will be excised differently. For example, some parts of the application may use the branch predictor more frequently than accessing L1 Instruction (I)-/Data (D)-cache. Similarly, if an application has a larger portion of load/store instructions, it spends most of its execution time in accessing the caches and/or memory for getting the data than the actual computation. Depending on the instruction mix, as mentioned in [49], applications can be broadly classified into three categories: (i) compute-intensive (ii) memory-intensive (iii) intermediate or mixed. The execution time (performance) of each category scales differently with operating frequency, mainly depends on the proportion of arithmetic, logical and memory operations. As reported in [20], MPEG4 decoding at 24 SVGA frames/s exhibits up to 7× workload variation measured in CPU cycles, due to decoding of intra-coded (I) SVGA frames with higher computations, followed by a number of predictive-coded (P) frames with lower computations [139]. In summary, it has been observed that the workload on the CPU varies depending upon the type of application and instruction mix during the execution [49]. Furthermore, the workload is also influenced by performance requirements of the application, such as frame rate in video processing applications, page loading rate for browsers, etc. [20].

It has been well studied that *V-f* setting can be adjusted depending on the workload on CPU to achieve energy savings [41, 50]. For example, Linux's ondemand power governor increases/decreases *V-f* level when the CPU workload (utilisation) is higher/lower than a predefined threshold [9]. However, such existing approaches are not effective for managing concurrent workloads, as they are not developed considering concurrent execution scenarios, contention on shared resources, and type of DVFS supported by a hardware platform (e.g., cluster-based DVFS, per-core DVFS, etc.). To understand and address the above problem, the effect of concurrent execution on CPU workload profile and application performance has to be investigated. Further, it is essential to devise a metric for classifying the workload in such scenarios to make appropriate *V-f* setting decisions. In the era of MPSoCs employing an increasing number of processing cores, the workload classification should be simple to minimize the overheads while providing necessary information to make runtime decisions. Therefore, an investigation of online workload classification is presented in the following sections.

3.1.1 Methodology

Application workload variations can be captured by the hardware PMCs or change in execution time with the frequency. Fig. 3.1 shows the proposed methodology for classifying the workload based on the statistical analysis of PMC data and execution time. It contains the following - An Odroid-XU3 hardware platform [28]; collection of execution time and PMC data for different execution scenarios; and statistical analysis of collected data for classifying the workload.

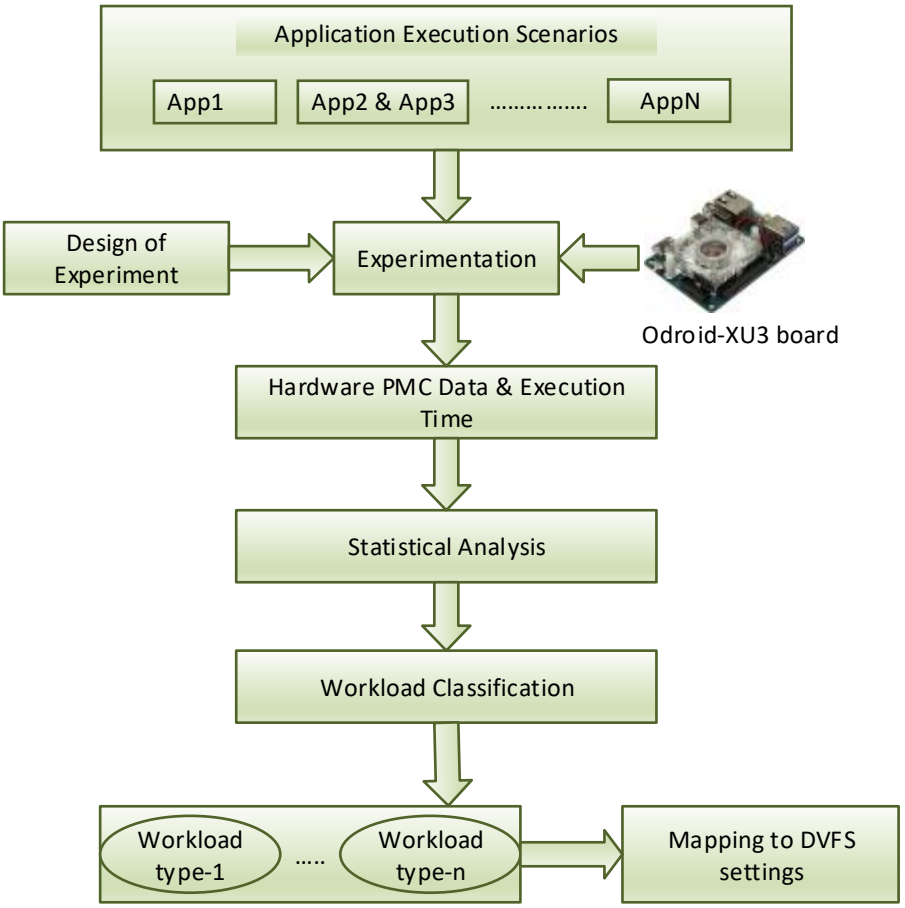

FIGURE 3.1: Experimental methodology used for workload classification

The application execution scenarios include sequential (individual) and concurrent execution of applications on the chosen hardware platform, the Odroid-XU3. This platform has been selected as it has the state-of-the-art mobile SoC with popular ARM big.LITTLE heterogeneous processing architecture, which is prevalent in tablet/smartphone devices). Moreover, as discussed in Section 2.4.1.1, it offers a wide variety of platform configurations such as core enabling/disabling, multiple DVFS settings per cluster and supports real-time sampling of power sensors and hardware PMCs. To understand the relationship between execution time and frequency for different application execution scenarios, the following experiments have been conducted. As part of this, `lbm`, `astar` and `mcf` applications from SPEC CPU2006 are considered [120], as these applications have significantly different instruction mix. Applications `lbm` and `mcf` have more number of memory operations than arithmetic operations, whereas `lbm` has higher number of arithmetic operations than memory operations. Each application is run on an ARM Cortex-A15 core and Cortex-A7 core of the Odroid-XU3 separately while sweeping the frequency from 200 MHz to 2.00 GHz on the Cortex-A15 core and 1.40 GHz 200 MHz on the Cortex-A7 core in 100 MHz steps. The execution time of an application for every frequency is measured by repeating the experiment ten times. Finally, for each frequency, the average execution time in seconds is computed by neglecting the execution

time in the first run to minimize the variation caused by inconsistent processor states, as per the recommendation by the ARM [140].

Similarly, the workload type also influences runtime information, i.e., hardware PMC data. Most of the modern processors support hardware PMCs, which can be sampled at runtime for various purposes, such as application profiling, debugging, runtime energy/performance management, etc. For example, ARM Cortex-A15 and Cortex-A7 cores support 66 and 42 performance events (e.g., cycle count, L1-data cache misses, etc.), respectively. However, only seven PMCs on Cortex-A15 and five PMCs on Cortex-A7 (including cycle count) can be sampled at a time without using time-division multiplexing. Although the collection of all events may give more information to accurately measure workload on CPU, this will have a significant impact on application performance and energy consumption due to runtime overheads [141]. Therefore, performance events with little correlation are selected using principal component analysis (PCA) [142] and variance-based ranking (VBR) [143], giving enough information to classify workload and select appropriate *V-f* setting.

Further, to identify the workload classes (e.g., memory-intensive, compute-intensive, etc.), k-means clustering is used due to its computational simplicity and efficiency [144]. The fundamental concept of cluster analysis is to form groups of similar objects as a means of distinguishing them from each other and can be applied in any discipline involving multivariate data [145]. The k-means algorithm owing to its computational simplicity is a popular clustering technique applied for a wide variety of applications [146]. It is a well-perceived fact in the research community that cluster analysis is primarily used for unsupervised learning where the class labels for the training data are not available [147]. With a given data set $X = \{x_i\}$, $i = 1, \ldots, n$ to be clustered into a set of k clusters, the k-means algorithm iterates to minimize the squared error between the empirical mean of a cluster and the individual data points, defined as the cost function (J),

$$J(\theta, u) = \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ij}(x_i - \theta_j)^2 \tag{3.1}$$

where θ_j is the cluster centre and $u_{ij} = 1$ if x_i lies close to θ_j, or 0 if otherwise [148]. Although k-means is a simple and efficient technique for clustering, it is sensitive to outliers, and its efficiency and number of iterations largely depends on seeds. A bad choice of initial centroids can have a great impact on performance. In some cases, it may converge to a local minimum. To surmount these issues, k-means is augmented with a simple seeding technique. Initial centroids have been chosen with probability proportional to the overall contribution. After defining initial centroids, the data vectors are assigned to a cluster label depending on how close they are to each centroid. The k centroids are recalculated from the newly defined clusters, and the process of reassignment of each data vector to each new centroid is repeated. The algorithm iterates over this

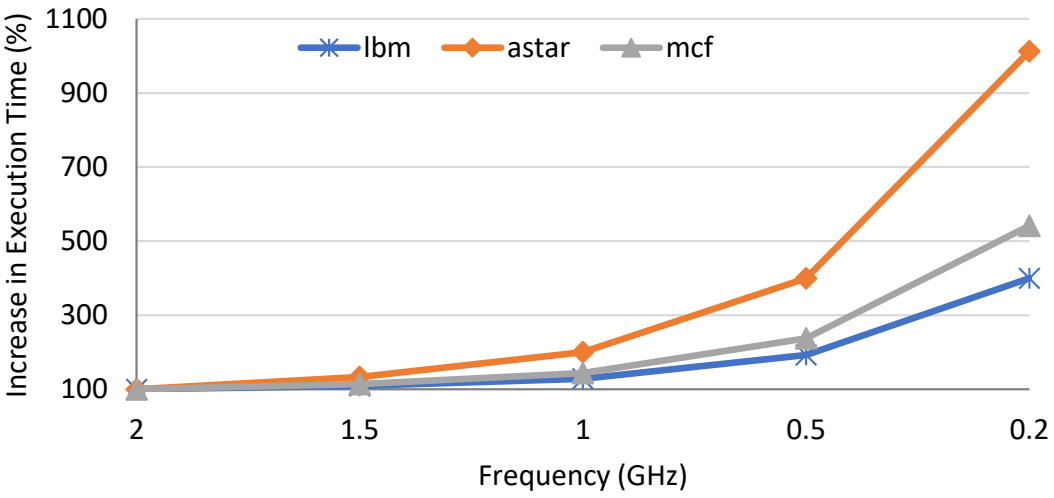

FIGURE 3.2: Variation in execution time with frequency for `lbm`, `astar` and `mcf` applications running on the Cortex-A15 core.

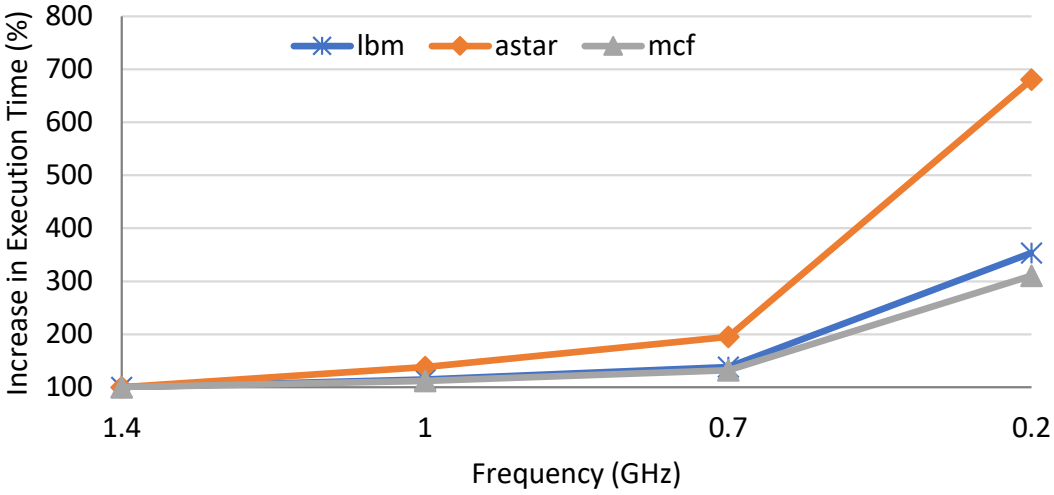

FIGURE 3.3: Variation in execution time with frequency for `lbm`, `astar` and `mcf` applications running on Cortex-A7 core.

loop until the data vectors from the dataset X form clusters and the cost function J is minimized [146].

3.1.2 Results and Discussion

This section presents the experimental results and analysis of workload classification using execution time and PMC data.

3.1.2.1 Execution Time Based Workload Classification

Fig. 3.2 and 3.3 show the variation in execution time for `lbm` `astar` and `mcf` applications when run individually. The relative percentage increase in execution time at a

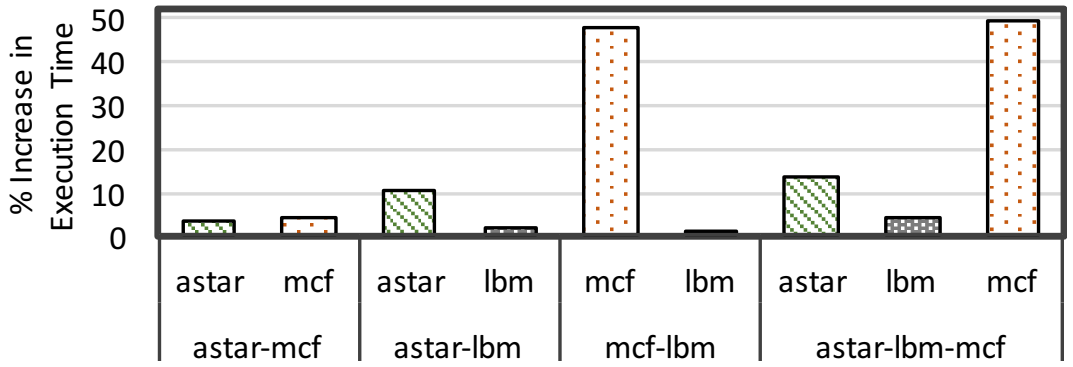

FIGURE 3.4: Effect of concurrent execution on application execution time.

particular frequency (`f`) is computed with respect to the execution time obtained at the maximum frequency ($\texttt{f}_{\texttt{max}}$), i.e., $\frac{\text{Execution Time@}\texttt{f}\times 100}{\text{Execution Time@}\texttt{f}_{\texttt{max}}}$. It can be observed that execution time of `astar` increases proportionally with decrease in operating frequency, whereas for `lbm` and `mcf`, variation is small till certain frequency (800 MHz on both Cortex-A15 and Cortex-A7 core). This is because of the high number of arithmetic and branch operations in `astar` compared to `lbm` and `mcf`, which have more memory (load and store) operations. Therefore, from the above, it can be understood that `lbm` and `mcf` are memory-intensive applications and `astar` is a compute-intensive application. When the operating frequency is lower than the memory speed (933 MHz), increase in execution time of all applications becomes prominent, including memory-intensive ones.

Moreover, the effect of concurrent execution on the execution time of individual application is also investigated. To do this, application scenarios `astar-lbm`, `astar-mcf`, `lbm-mcf` and `astar-lbm-mcf` are considered for the analysis. Each application is allocated to a particular core in the A15-cluster. Then, the execution time of each application in all combinations is compared with its execution time when run individually. Our observation shows a change in the execution time of each application when run concurrently with other applications. This change can be attributed to contention due to simultaneous access to shared L2 cache/memory by each application for getting the data when all concurrent applications just start their execution. Therefore, if the data size of each application can fit into the L1 cache of its allocated core, the change in execution time is insignificant. However, if the selected application has frequent access to shared L2 cache/memory, there will be a significant variation in execution time. As shown in Fig. 3.4, execution time of each application gets increased (up to 50%) due to contention on memory. The effect on application execution time becomes more prominent when the number of concurrent applications, especially memory-intensive ones, increases. This necessitates an efficient online classification of concurrent workloads to understand the above effects and select an appropriate *V-f* setting for achieving energy efficiency.

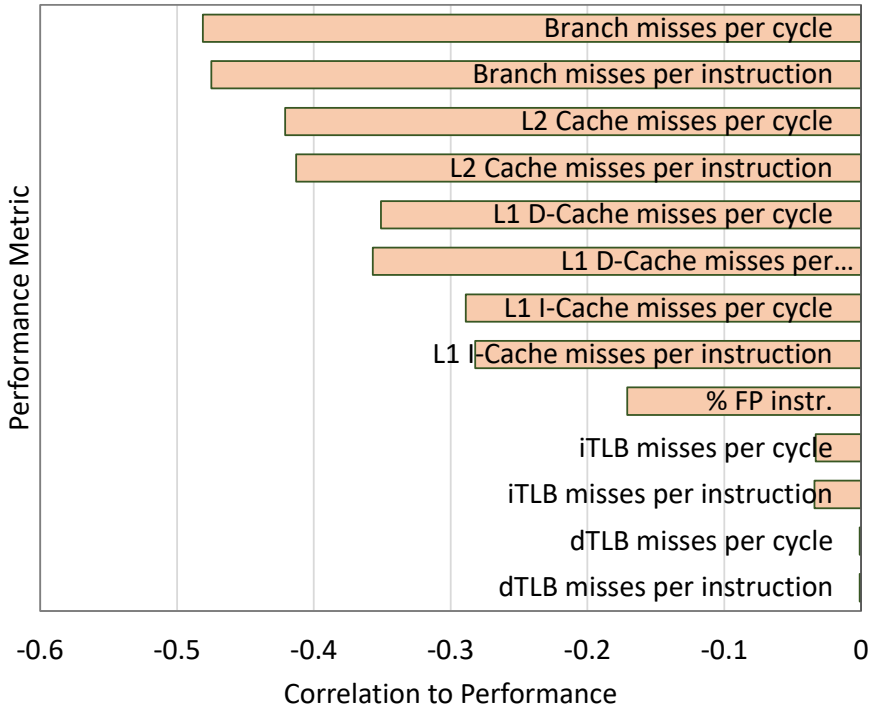

FIGURE 3.5: Correlation between different performance metrics and application performance, measured in IPC.

3.1.2.2 PMC Data Based Workload Classification

The aforementioned execution time-based analysis helps in selecting memory- and compute-intensive applications and understanding the variation in execution time when multiple applications execute concurrently. However, it is not effective for runtime characterisation of workload and selecting an appropriate *V-f* level, as it needs executing the application at different frequencies, thereby leading to significant runtime overheads.

To identify the most appropriate metric using the top performance events reported by the PCA and VBR, Pearson's correlation coefficient (ranges between -1 and +1) is obtained for the metrics computed from PMCs and instruction per cycle (IPC). Fig. 3.5 shows the average correlation between various performance metrics and IPC. Note that the negative correlation in the figure shows that the increase in selected metrics results in decreased performance. The figure demonstrates that branch misprediction and cache miss events have a high correlation with IPC. Performance events such as branch mispredictions and L1 I-/D-Cache misses involve in-core activity and thus do not benefit from frequency scaling. A similar study presented in [49], a successful way to DVFS is to account only for the stall CPU cycles introduced by the off-chip non-overlapping misses, i.e., Last Level Cache (LLC) misses, which is L2 Cache on the Odroid-XU3. DVFS approach can capitalise on the LLC misses by appropriately tuning the DVFS setting. As an example, L2 cache misses (LLC misses) and cycle count have been collected for every 200 ms for two execution scenarios: executing individually

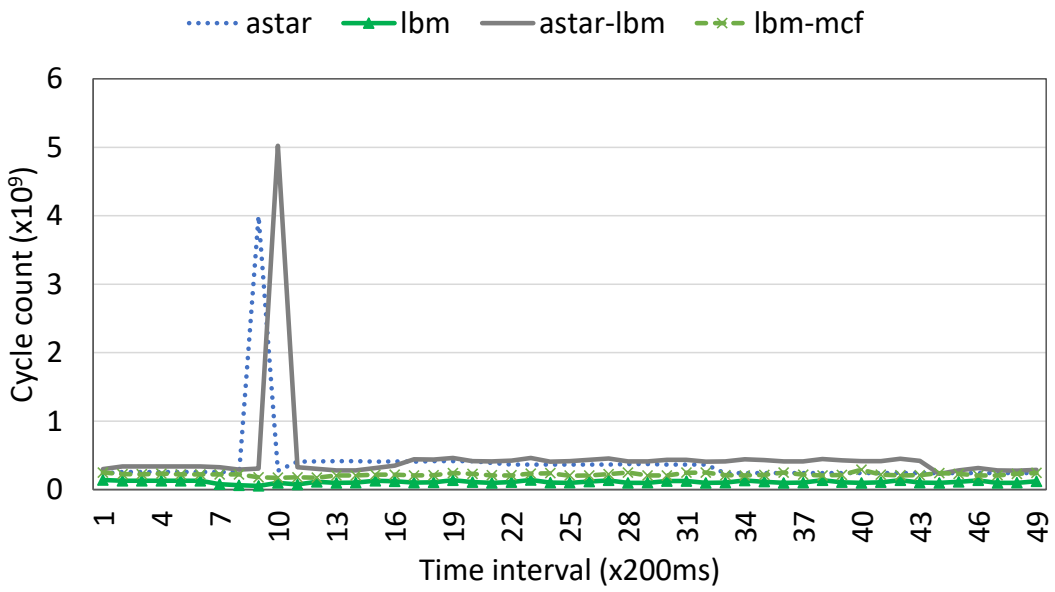

FIGURE 3.6: Cycle count for sequential and concurrent execution of `astar`, `lbm` and `mcf`. Samples are collected for every 200 ms.

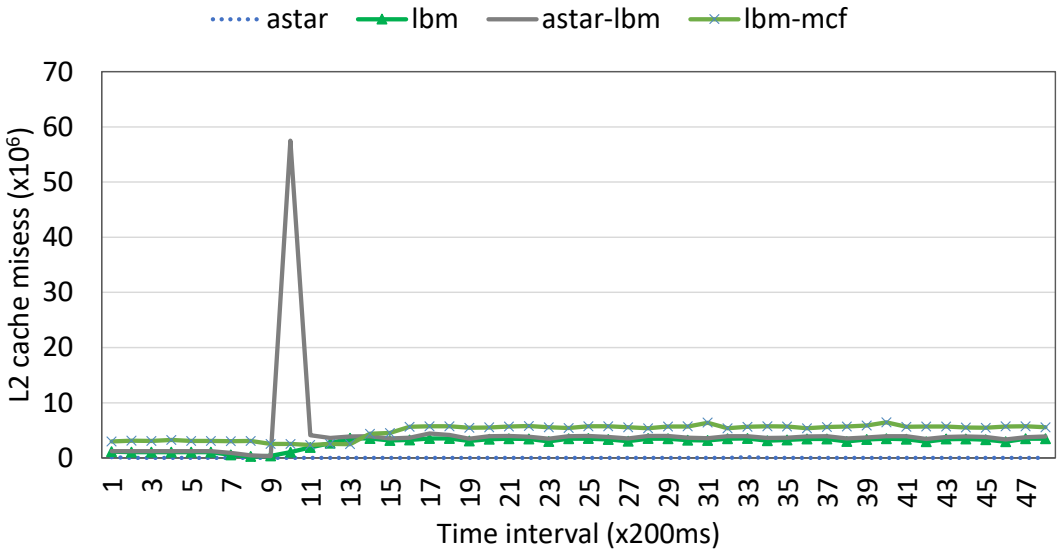

FIGURE 3.7: L2 cache misses for individual and concurrent execution of `astar`, `lbm` and `mcf`. Samples are collected for every 200 ms.

and concurrently. Fig. 3.6 and Fig. 3.7 show the cycle count and L2 cache misses for application scenarios: `astar`, `lbm`, `astar-lbm`, and `lbm-mcf`, having a mix of compute- and memory-intensive applications. It can be observed that the average number of L2 cache misses of `astar` are substantially small compared to `lbm` or a concurrent execution scenario having at least one memory-intensive application (e.g., astar-lbm). On the other hand, the average cycle count of `lbm` is significantly less compared to `astar` and other application scenarios. Moreover, for concurrent execution of memory- and compute-intensive applications, the cycle count is mainly influenced by compute-intensive, and L2 cache read refills are mostly affected by the memory-intensive workload.

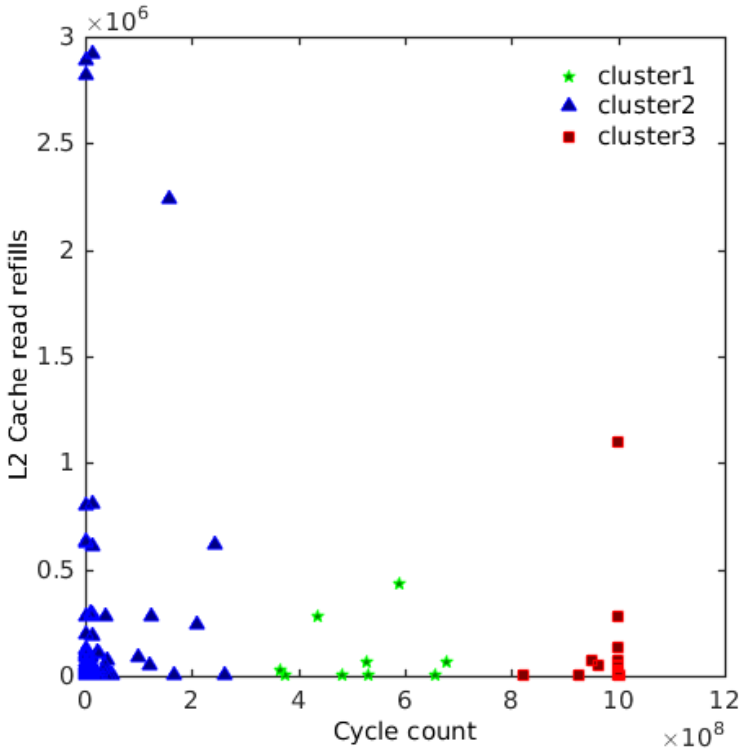

FIGURE 3.8: K-means clustering of workloads for concurrent execution of `astar` and `lbm`.

Fig. 3.8 shows k-means clustering of workloads for parallel execution of `astar` and `lbm`. Each cluster in the figure represents a workload class. Even though `astar` and `lbm` are classified as compute- and memory-intensive respectively, based on PMC data analysis, three or more workload classes can be formed by changing the cluster size, shown in the figure. It signifies that when two applications with distinct workload characteristics are executed concurrently, more than two workload classes (compute-intensive, memory-intensive, mixed, etc.) could be seen. This implies that if a multi-core platform supports a fine-grained control of DVFS, each workload class can be assigned to an appropriate DVFS setting, leading to lower power consumption with a negligible performance loss. As per the above discussion, it can be understood that the number of LLC misses, equivalent to memory accesses, can measure the degree of memory-intensiveness of an application. During execution of an application, workload variations have to be captured at regular intervals to select an appropriate DVFS level to improve energy efficiency. Therefore, to classify the workload of an application during its execution, the following two metrics are derived using LLC misses:

1. Memory Accesses Per Cycle (MAPC)
2. Memory Accesses Per Instruction (MAPI)

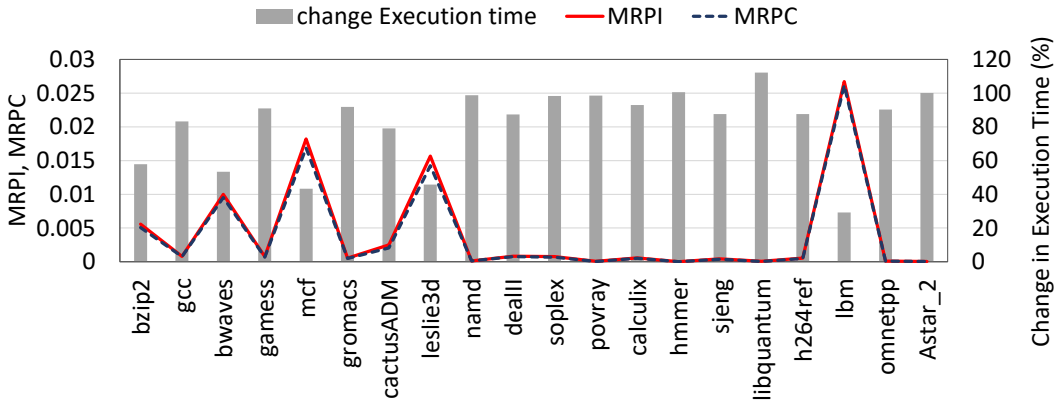

FIGURE 3.9: MRPI and MRPC of various applications, computed by executing each application to completion. The change in execution time when frequency is scaled from 2 GHz to 1 GHz.

MAPC and MAPI are calculated by dividing the number of LLC misses with cycle count and instructions retired, respectively. The Odroid-XU3 supports measuring of only memory reads (does not include memory writes), called L2 cache read refills. Therefore, similar to MAPC and MAPI, the following metrics are derived: Memory Reads Per Cycle (MRPC), and Memory Reads Per Instruction (MRPI). To show that MRPI and MRPI are appropriate metrics for workload classification, the following experiment is conducted over a set of applications from SPEC CPU2006. For each application, MRPI, MRPC and execution time are measured with the help of hardware PMCs for two different frequencies 2 GHz and 1 GHz. As discussed before, the execution time of compute-intensive applications increase greatly compared to memory-intensive ones when the frequency is scaled down. The similar behavior can be seen in Fig. 3.9, where some applications show nearly 100% increase in their execution time when the frequency is raised by 100%. This kind of workload characteristics has been well captured by MRPI and MRPC; A highly compute-intensive workload on the processing core indicates a low MRPI or MRPC and vice versa. Even though, both MRPI and MRPC metrics help in deciding the workload behavior at runtime, MRPI is selected over MRPC because of its frequency agnostic behavior compared to MRPC. Such a feature of MRPI facilitates to capture the workload changes during execution time well even when frequency varies, which is essential to select an appropriate DVFS setting.

Furthermore, as demonstrated in the following section, MRPI based energy management approaches perform better compared to the commonly used CPU cycles, Instructions Per Cycles (IPC), or utilisation [9, 149, 150]. Note that CPU cycles, utilisation, and IPC are usually affected not only by memory reads but also by other statistics, such as lower-level cache misses, branch mispredictions, etc., which involve only in-core operations. The penalty (measured in cycles) for doing extra in-core operations remains the same no matter what the frequency is [151]. However, in the case of multi-threaded applications, the CPU workload also gets affected by the thread synchronisation contention. Therefore, in such cases, Section 3.3 shows that metrics such as utilisation

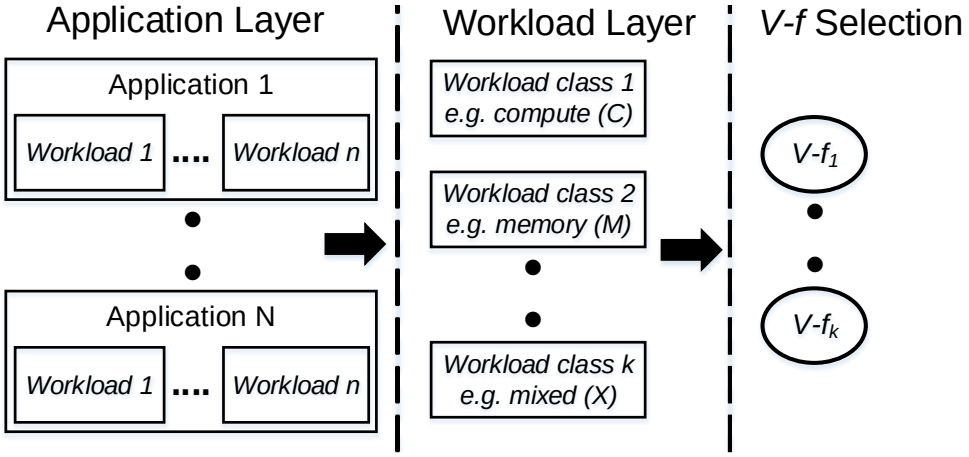

FIGURE 3.10: Conceptual overview: Concurrent applications with associated workload classes and the need for optimal *V-f* selection.

indicating in-core activity are important for runtime power management.

3.2 Runtime Management of DVFS for Concurrent Execution of Single-threaded Applications

Fig. 3.10 presents a conceptual overview of multiple applications undergoing concurrent execution, having different workload classes, such as compute-intensive, memory-intensive and mixed [49, 134] resulting from the varying means they exercise the hardware. From a performance perspective, it is desirable to run a compute-intensive application at a higher clock frequency as compared to memory-intensive one. Hence, selecting the appropriate *V-f* for the associated workloads is key to achieving the desired energy-performance trade-off.

Energy efficiency and high performance requirements have been a key research focus of processor designers in multi-core platforms [34]. These systems are equipped with dynamic voltage and frequency scaling (DVFS) which enables on-the-fly linear reduction of frequency (f) and voltage (V), yielding a cubic reduction in dynamic power consumption ($\propto V^2 f$). As discussed in Chapter 2, energy can only be saved if the power consumption is reduced enough to cover the extra time it takes to run the workload at a lower Voltage-frequency (*V-f*). To reduce the complexity of hardware design and implementation, modern embedded multi-core architectures with a fixed number of cores are organized as clusters, where all cores in a cluster operate at the same *V-f* (cluster-wide DVFS) [152]. For example, popular hardware platforms such as the Odroid-XU3 [28], MediaTek helio X20 [30] and Juno r2 [29] contain such an architecture.

Modern systems usually need to execute multiple applications concurrently, where each application exercises the hardware differently based on its instruction mix (e.g., 70% load/store and 30% arithmetic). Moreover, these applications share system resources,

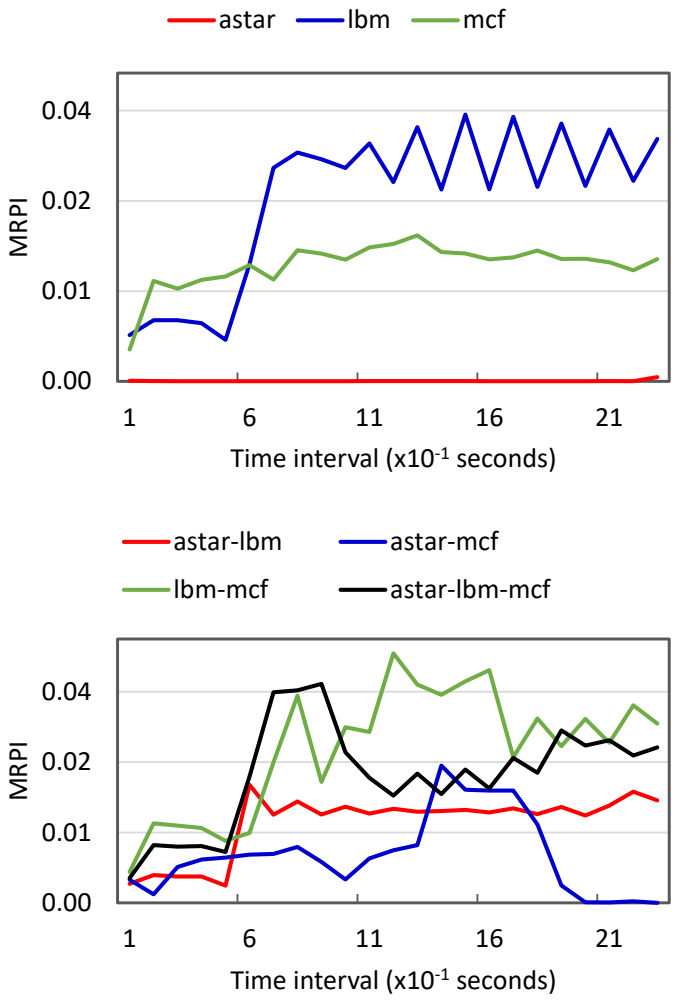

FIGURE 3.11: Variation in MRPI for individual (top) and concurrent (bottom) execution of applications.

resulting in varying and mixed workloads (compute-intensive, memory-intensive, etc.). Managing such workload variations on multi-core architectures supporting cluster-wide DVFS is a challenging task, as each core may execute a different type of workload, and thus an appropriate *V-f* for the entire cluster has to be determined. From a performance perspective, it is desirable to run a compute-intensive application at a higher *V-f* as compared to memory-intensive one. Hence, selecting the appropriate *V-f* setting for the associated workload is key to achieving the desired energy-performance trade-off.

In case of concurrent execution, there is not a marked distinction between compute- and memory-intensive workloads due to resource sharing (e.g., last level cache and memory). Fig. 3.11 demonstrates the variation in workloads when multiple applications are run in two different configurations - individually (top) and concurrently (bottom) on the A15 cluster of the Odroid-XU3 platform. Here, three applications, having different workloads from SPEC CPU 2006, are considered: *astar*, *lbm* and *mcf*, and their various combinations *astar-lbm*, *astar-mcf*, *lbm-mcf* and *astar-lbm-mcf*.

It can be observed from Fig. 3.11 that the different workload types (of the applications astar, lbm and mcf) can clearly be classified when run individually, which is completely different in the case of concurrent execution having greater workload variability.

Moreover, the execution time of each application gets increased due to contention on memory and it becomes more prominent when the number of concurrent applications, especially memory-intensive ones, increases. This necessitates an efficient online classification of concurrent workloads to understand the above effects and select an appropriate *V-f* setting for achieving energy efficiency.

A close observation of existing approaches [19, 49, 50, 153] (detailed in Chapter 2) shows that they are not efficient for online energy management of concurrently executing applications on a cluster-based multi-core architecture. In order to overcome the limitations of existing approaches, this work makes the following contributions:

1. An approach for workload selection and prediction that takes memory contention into account for pro-active control of *V-f* setting.
2. Online concurrent workload classification using MRPI to efficiently select *V-f* settings based on the predicted workload.
3. A light weight technique for identifying cores that are not executing any application (unused cores) to avoid selection of a high *V-f* value, as unused cores usually have low MRPI.
4. Implementation and validation of the proposed approach on a real hardware platform, the Odroid-XU3 [28].

3.2.1 Runtime Management of DVFS

The proposed approach addresses the following problem:

Given a set of concurrent applications and a multi-core platform supporting cluster-wide DVFS
Optimize energy consumption while maximizing the performance by selecting efficient *V-f*

A multi-core system can be represented in three-layers as shown in Fig. 3.12 (left). These layers interact with each other to execute an application, indicated by arrows. The top most layer is the application layer, which is composed of multiple applications, having different workload profiles. The operating system layer, shown in the middle (e.g., iOS, Linux, etc.), coordinates the execution of an application on the hardware (bottom layer), consisting of multi-core processor. An overview of the proposed approach is shown in Fig. 3.12 (right), having the following stages:

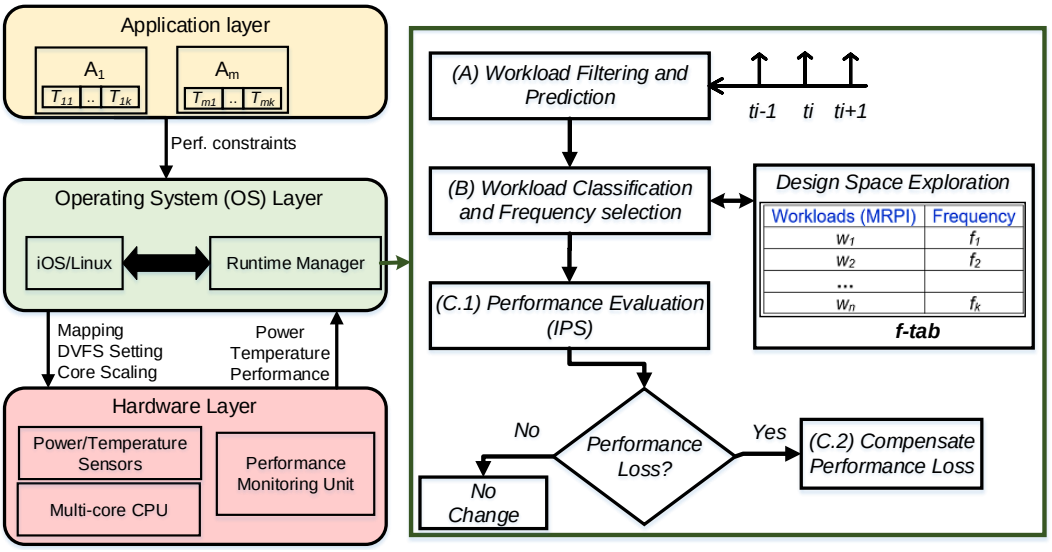

FIGURE 3.12: Representing a multi-core system as three stacked layers: application, operating system and hardware (left), and overview of the proposed online energy minimization technique (right).

(A) workload selection and prediction (Section 3.2.1.1)

(B) workload classification and frequency selection (Section 3.2.1.4)

(C) performance evaluation and compensation (Section 3.2.1.6)

A detailed discussion on each stage is presented in the following sections, with pseudo code shown in Algorithm 1.

3.2.1.1 Workload Selection and Prediction

An appropriate *V-f* setting depends on the application workload. When applications execute individually, *V-f* setting is mostly guided by a single workload [149]. However, for concurrent execution, it depends on multiple workloads. Therefore, *V-f* setting at time t_i can be represented as,

$$V\text{-}f_i = \begin{cases} f\ (w_i) & \text{Individual workload} \\ f\ (w_{1i}, w_{2i}, \ldots, w_{Ni}) & \text{Concurrent workloads} \end{cases} \tag{3.2}$$

where, 1 to N represent applications. It is important to note that, for concurrently executing applications on a cluster-based multi-core, *V-f* of each cluster should be chosen in such a way that all the applications meet their performance requirements. Furthermore, these applications generate varying and mixed workloads due to resource sharing (e.g., L2 cache and memory) showing intra and inter workload variations during their execution (see Fig. 3.11). Therefore, a representative *V-f* setting has to be chosen to

guide the further stages for achieving energy efficiency without degrading application performance.

3.2.1.2 Workload Selection

Let us assume that there are N concurrently executing applications and w_{ni} be the workload of an application n for time interval $t_{i-1} \rightarrow t_i$. There will be N different workloads at every time interval of the execution. The workload is quantified by the MRPI, where a low value represents a high load on the processing core and vice versa. Due to limited resource availability, concurrent execution of applications creates contention on shared resources, especially on memory, impacting performance of individual application, i.e. increases the execution time.

The increase in execution time (latency) of each application is calculated at runtime based on the average memory-intensiveness of the running applications. We represent this latency as δ amount of increase in the workload. If all the running applications are memory-intensive, then the value of δ will be high due to increased memory traffic. The value of δ will be used to scale down the frequency further to minimize the wasted cycles/switching activity. To achieve maximum performance for each application, *V-f* setting for time interval $t_{i-1} \rightarrow t_i$ (considering cluster-wide DVFS) is influenced by the workload,

$$w_{targeti} = \min\{w_{1i}, w_{2i}, w_{3i}, w_{4i}, ..., w_{Ni}\} + \delta_i \tag{3.3}$$

3.2.1.3 Workload Prediction

To adapt to workload variations and to achieve energy minimization, proactive control of *V-f* is of utmost importance. Therefore, the future workload (t_{i+1}) needs to be predicted at t_i to set the appropriate *V-f* value for the time interval $t_i \rightarrow t_{i+1}$. To accomplish this, an exponential weighted moving average (EWMA) filter [101] is used to predict workload p_{i+1} during the interval $t_i \rightarrow t_{i+1}$ (line 12, Algorithm 1),

$$p_{i+1} = \gamma \times a_i + (1 - \gamma) \times p_i \tag{3.4}$$

where $\gamma \in [0,\ 1]$, p_i and a_i are the smoothing factor, predicted and actual workloads respectively during the interval $t_{i-1} \rightarrow t_i$. It is to be noted that $w_{targeti}$ computed from Equation 3.3 represents the actual workload a_i. To minimize workload miss-predictions,

Algorithm 1 Online energy minimization approach

Input: T_s, profile data (f-tab) and l_{as}
Output: Frequency

```
1: PMU_initialize()
2: f_cur = cpufreq_get_frequency (core#)
3: p_w ← 0; W_c ← 0
4: run for T_s at f_max and compute IPS_max
5: while (1) do
6:     if (W_c ≠ l_as) then
7:         wait for T_s
8:         compute new IPS value (IPS_n)
9:         actual workload (a) = get_workload_pmcdata()
10:        prediction error (P_e) = a − p
11:        γ = α × P_e + β
12:        EWMA (a, p, γ, P_e) (Equation 3.4 & 3.5)
13:        compute memory latency (δ), classify workload, and get f_new from f-tab
14:        if (f_new ≠ f_cur) then
15:            cpufreq_set_frequency (core#, f_new)
16:            f_cur ← f_new
17:            W_c ← W_c − 1
18:        else
19:            W_c ← W_c + 1
20:        end if
21:        perf_loss (λ) ← ((IPS_max − IPS_n)/IPS_max) × 100
22:        if (λ > Δ%) then
23:            f_new ← f_new + λ × f_max
24:        end if
25:    else
26:        wait for T_s × l_as
27:        W_c ← 0
28:    end if
29: end while
30: function GET_WORKLOAD_PMCDATA()
31:    collect PMC data and set MRPI of unused cores to 10
32:    return the selected workload (Equation 3.3)
33: end function
34: PMU_terminate()
```

the predicted workload of the interval $t_{i-1} \rightarrow t_i$ is compared to the actual workload measured from hardware PMCs. Subsequently, computed prediction error P_e (difference between actual and predicted workloads) is used to improve the workload prediction for $t_i \rightarrow t_{i+1}$. The accuracy of workload prediction highly depends on γ and a fixed value of γ would result in frequent miss-predictions, if there are large workload variations. Therefore, the value of γ is changed in proportion to p_{wc} ($=P_e/p_i$),

$$\gamma = \alpha \times P_{wc} + \beta \tag{3.5}$$

3.2.1.4 Workload Classification and Frequency Selection

Classification of the predicted workload is important for identifying an appropriate *V-f* setting for achieving energy savings without any performance loss. For online workload classification, hardware PMCs are used for periodically getting performance statistics (L2-cache misses and instructions retired) during application execution. The modified performance monitoring tool `perfmon` [14] is used for accessing the PMCs, initialized and terminated through `PMU_initialize` and `PMU_terminate` routines (lines 1 & 34, Algorithm 1).

To minimize the runtime overhead, workload types are predetermined through a custom program, generating varying the number of memory accesses. The custom program copies varying amount of data from one memory location to another, like Memcopy [154], after doing addition and multiplication operations on two large arrays. At different number of memory accesses, the variation in MRPI and execution time is recorded by sweeping the frequency from 0.2 GHz to 2 GHz on A15 cluster of Odroid-XU3. The offline profiling results contain MRPI ranges and corresponding appropriate *V-f* settings, which are used at runtime to set the operating frequency to a desired value through the utility `cpufreq-set` (lines 13 & 15, Algorithm 1).

The range of MRPI values having little (<1%) or no effect on execution time for the same frequency are grouped into a single class (same workload type). Application execution intervals with large MRPI are actually memory-intensive workloads, so they can be run at lower frequencies to save energy as higher frequencies will simply result in wasted CPU cycles while waiting for data from memory. This motivates us to assign a low frequency to large MRPI workloads and it is decided by the speed of memory (933 MHz in the present case). Similarly, workload having a significantly low MRPI, i.e. execution time scales linearly with frequency, is assigned a maximum available frequency (2 GHz on A15 cluster).

Identification of unused cores

MRPI of unused cores, i.e. no application is executing on that core, is usually low due to fewer memory reads. As a result, if the unused cores are not identified, it gives a miss-impression that the cores are executing a compute-intensive application, leading to selection of a high *V-f* value for whole cluster and thus increasing energy consumption. This becomes prominent when there are more cores than number of concurrent applications in a cluster. To address this, the proposed algorithm determines unused cores at runtime using an IPS threshold. If IPS of a core is below the threshold value, it is identified as unused core and subsequently, its MRPI is set to 10 (any value larger than one would be fine as the value of MRPI usually does not exceed one). This eliminates the influence of unused cores on *V-f* setting, which is decided by the minimum MRPI

of the applications (Equation 3.3). The IPS threshold value is experimentally identified as 2.9×10^6 for the cores in A15 cluster of Odroid-XU3, which is used for experimental validation. Determining unused cores using IPS does not need extra PMCs as the number of instructions retired during each time interval is already made available for computing MRPI. Otherwise, checking unused status with generally used CPU utilisation [150] needs an extra PMC (number of active CPU cycles), leading to increased runtime overhead.

3.2.1.5 Adaptive Sampling

In general, not all applications exhibit the frequent workload variations during the execution. For example, in Fig. 3.11, *astar*, *mcf* and *astar-lbm* show significantly less workload variations compared to *lbm* and *lbm-mcf*. In that case, performing PMC data collection, subsequent processing and *V-f* switching (Algorithm 1) periodically (at every T_s) would lead to unnecessary runtime overheads without any benefits. Therefore, to avoid this, the sampling interval is adjusted according to the application workload variations. The proposed algorithm keeps track of workload variations through a counter W_c which gets incremented/decremented if *V-f* setting for time intervals $t_{i-1} \rightarrow t_i$ and $t_i \rightarrow t_{i-1}$ is different/same. When the value of W_c reaches l_{as}, a configurable parameter, the runtime adaptation is paused for $l_{as}\times T_s$. This stops unnecessary PMC data collection, subsequent processing and *V-f* switching.

3.2.1.6 Performance Evaluation and Compensation

Considering the dynamic resource availability and interference between concurrent applications, it is important to evaluate the performance during execution to ensure that no application experiences any performance loss. Therefore, instructions per second (IPS) is considered as a metric for quantifying the runtime performance of each application for every elapsed time interval T_s. The performance loss is calculated by comparing the IPS_n for every time interval with the maximum IPS (IPS_{max}) achieved by executing the application at the highest available frequency (f_{max}) (line 4, Algorithm 1). If there is a performance loss of λ% during the interval $t_{i-1} \rightarrow t_i$, the selected *V-f* will be increased by $\lambda\times f_{max}$ (line 23, Algorithm 1) for subsequent time interval ($t_i \rightarrow t_{i+1}$) to compensate it. Furthermore, the frequency is modified only when the performance loss (λ) is significant to minimize the overheads associated with DVFS [99]. We experimentally verified and set the value of λ to 1% by taking the variations in PMC data into account.

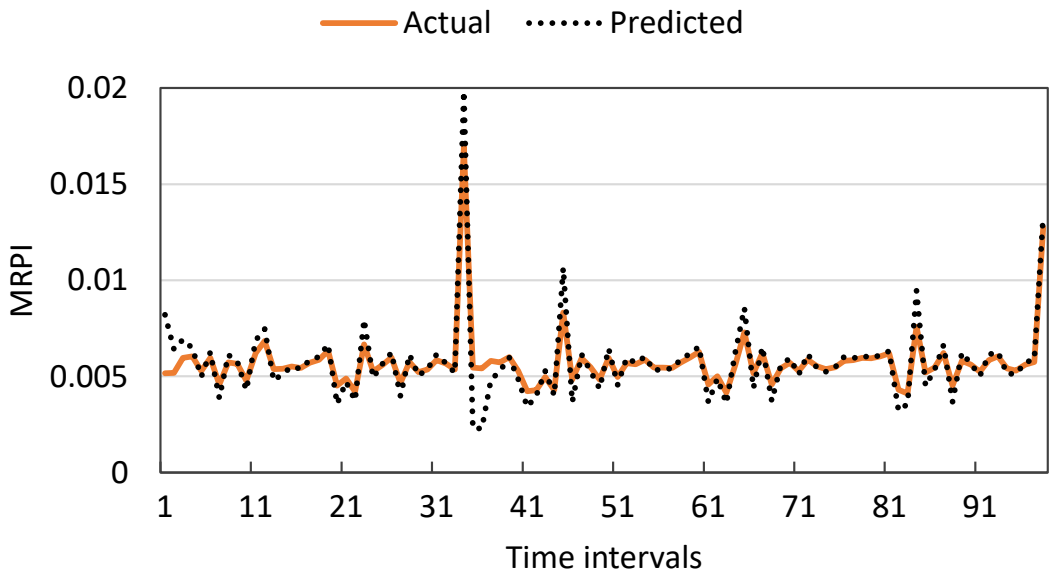

FIGURE 3.13: Workload prediction for the application scenario *mi-mc*.

3.2.2 Experimental Results

The proposed online energy optimization technique is validated on the big cluster of the Odroid-XU3 platform running Ubuntu Linux Kernel 3.10.96. To show the effectiveness of the proposed approach, applications *lbm* (lb), *milc* (mi), *mcf* (mc) and *bwaves* (bw) from SPEC CPU2006 [120], and *swaptions* (sw) and *freqmine* (fr) from PARSEC [15] are considered. Details of platform and benchmark are given in Section 2.4.1 and 2.4.2 respectively. Applications are executed concurrently in single, double and triple combinations. The power is measured from the on-board power sensors of Odroid-XU3 every 100 ms. The experimental results are collected by running each scenario ten times and finally, their average values (energy and performance) are computed for the comparison. The proposed technique is compared against Linux's conservative, ondemand and interactive frequency governors, which are implemented on millions of smartphones, making them competitive baselines [155]. These governors come under utilisation-based approaches as they scale the frequency based on utilisation threshold (more details are presented in Section 2.3.1). Further, IPC-based [150] and exhaustive search-based, similar to [50], approaches are also considered for the comparison. For exhaustive search-based approach, each application scenario is executed at all available frequencies and the one with minimum energy consumption is selected while having the same or better performance than the proposed approach.

3.2.2.1 Workload Prediction and Adaptive Sampling

The values of α, β in Equation 3.5 were experimentally obtained by sweeping them between 0 and 1, and observing the corresponding workload miss-predictions (under/over)

for various application scenarios. Finally, a value of 0.3 and 0.6 are chosen as it resulted in relatively accurate workload prediction. Fig. 3.13 shows the actual and predicted MRPI for the application scenario *mi-mc* at different time intervals. The average error in workload prediction, considering all application scenarios used in evaluation, is 4.2%.

The value of sampling interval (T_s), for which a new value of *V-f* is computed and set, is determined by varying its value from 20 ms to 250 ms in 20 ms steps and identified that 200 ms is good in terms of energy savings and runtime overheads. Further, the effect of adaptive sampling for various application scenarios is evaluated, where improvement of up to 1.83% and 1.62% in energy consumption and execution time are observed, respectively.

3.2.2.2 Energy Savings

In single application scenario, only one applications is active at a given moment of time. Fig. 3.14 (a) shows energy consumption values for various approaches, executing one application at a time. Proposed approach is able to outperform the reported techniques by showing energy savings of up to 68%, 66%, 65%, 63%, and 39% compared to interactive, ondemand, conservative, IPC-based and exhaustive search-based approaches, respectively.

At a given moment of time, multi-core platforms may need to execute applications concurrently. Such scenarios usually observed in smartphones, where an user tries to run multiple applications at the same time, e.g., MP3 player and Facebook. To validate such scenarios, concurrent execution of double and triple applications is considered. The proposed approach efficiently adapts to concurrent workload variations compared to existing techniques and selects an appropriate *V-f* setting (see Fig. 3.15). Further, it also considers latency (δ) due to memory contention, whose value is experimentally identified as 4.5% of average MRPI of all cores in a cluster. For double application scenario, proposed technique improves energy savings by up to 69%, 66%, 65%, 65% and 54% compared to interactive, ondemand, conservative, and IPC-based and exhaustive search-based approaches, respectively. Furthermore, for triple application scenario, proposed approach consumes 67%, 65%, 65%, 63%, and 35% less energy than interactive, ondemand, conservative, IPC-based and exhaustive search-based approaches, respectively. As can be seen from Fig. 3.15, utilisation-based approach chooses the highest frequency even though the workload is memory-intensive. It is because of the fact that while the workload is memory intensive, it places a high load on the cores as far as the load measured by the kernel is concerned. As a result, utilisation-based approach runs at the highest frequency even if it does not improve the performance.

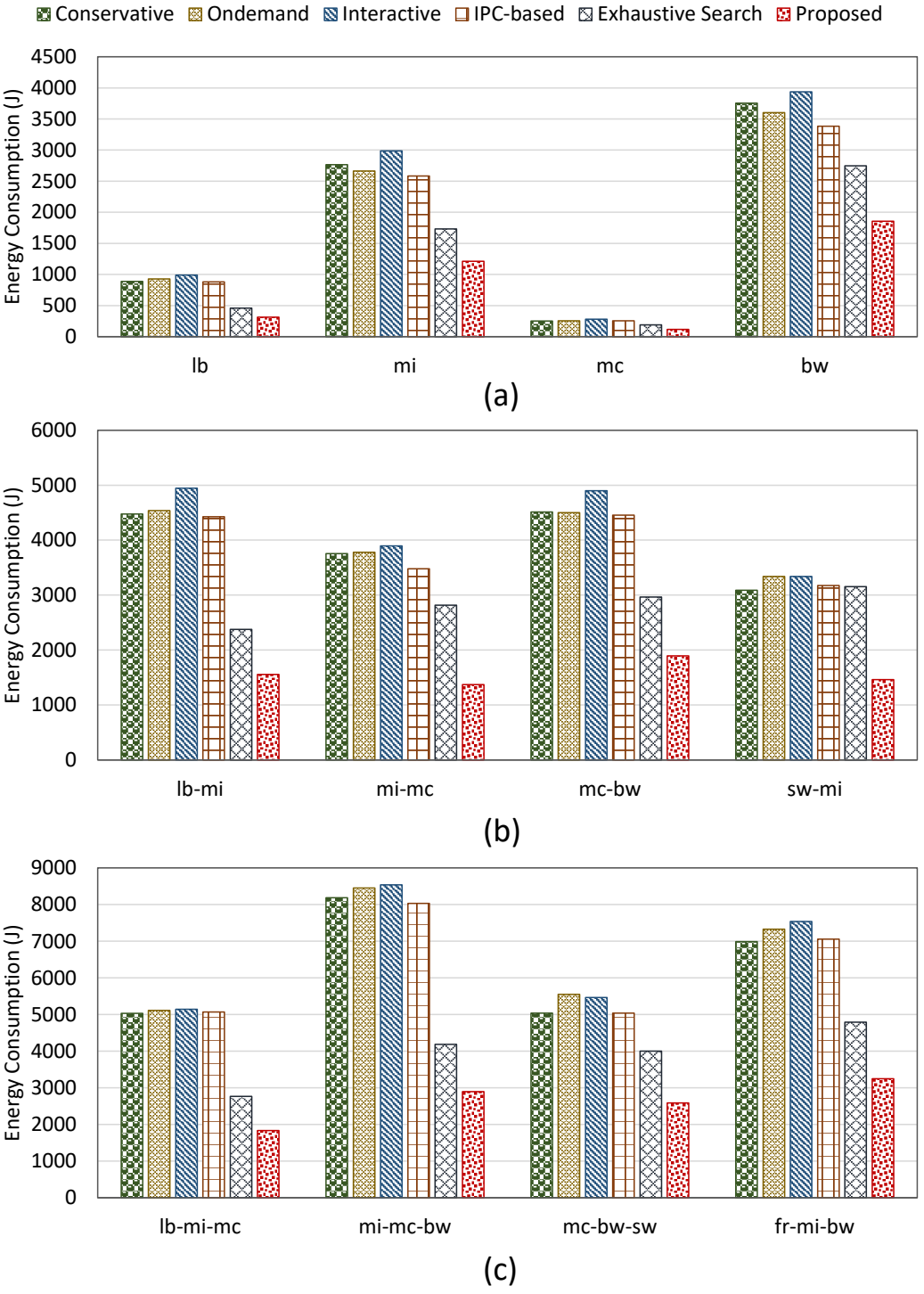

FIGURE 3.14: Comparison of proposed approach in terms of energy consumption with the existing approaches for different execution scenarios– a) single-application, b) double-application, and c) triple-application.

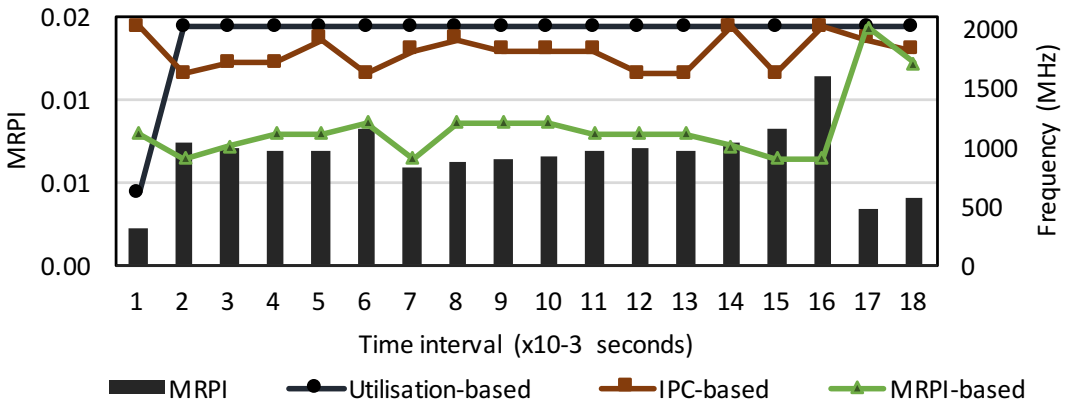

FIGURE 3.15: MRPI and frequency at different time intervals of the application scenario *lb-mi* execution for various approaches.

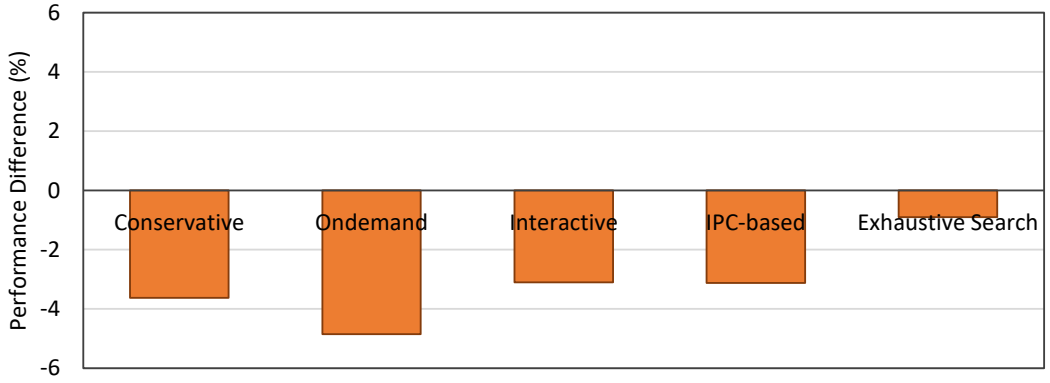

FIGURE 3.16: Average performance difference between proposed approach and other reported approaches.

3.2.2.3 Application Performance

The application performance is evaluated for various application scenarios by computing the average execution time over several runs. The proposed technique continuously monitors the application performance in terms of IPS for every time interval and consequently, modifies the chosen frequency if there is a significant performance loss ($>$ 1%). As a result, the proposed approach achieves significantly better energy savings with a little performance loss. As shown in Fig. 3.16, the average execution time for the proposed approach, considering different application scenarios, is 4.85%, 3.62%, 3.10%, 3.22% and 0.9% slower compared to interactive, conservative, ondemand, IPC-based and exhaustive search-based approaches, respectively.

3.2.2.4 Runtime Overheads

The runtime overhead associated with the proposed approach can be represented as,

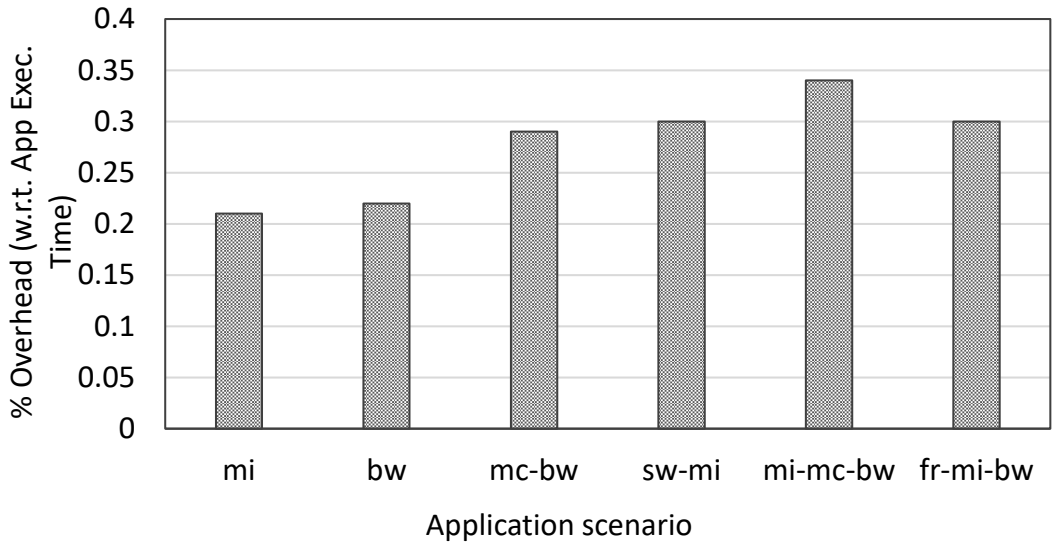

FIGURE 3.17: Runtime overhead of the proposed approach.

$$T_o = \frac{T_{exec} - r \times l_{as} \times T_s}{T_s} \times [S_o + C_o + P_o] \tag{3.6}$$

where T_o, T_{exec}, r, S_o, C_o, and P_o, represent total overhead; execution time; number of times the adaptation is paused; overheads associated with *V-f* switching; PMC collection and remaining processing steps (involving l_{as} and T_s) shown in Algorithm 1, respectively. S_o and C_o are platform dependent, which takes 300 ns and 30 ns per counter for the Odroid-XU3 [50]. Fig. 3.17 shows the runtime overhead of the proposed technique for various application scenarios as a percentage of application execution time. A maximum overhead of 0.35% is observed for *mi-mc-bw*, having a long execution time of 111 seconds. This shows that proposed approach has negligible runtime overhead.

3.2.3 Summary

An online energy minimization approach for concurrently executing single-threaded applications on a homogeneous multi-core platform is presented. It uses workload classification and prediction for appropriately selecting the *V-f* setting at runtime. Validation on a real multi-core hardware platform for various application scenarios shows an improvement of up to 69% in energy efficiency compared to existing approaches. Furthermore, it incurs a runtime overhead of up to 0.35% of application execution time, which is relatively small.

3.3 Runtime Management of DVFS for Concurrent Execution of Multi-Threaded Applications

There have been various approaches proposed to improve the energy efficiency for multi-threaded applications [109–115, 118, 119, 156–159] using low-power techniques such as dynamic voltage and frequency scaling (DVFS), power capping or clock gating (please refer to Section 2.3.2 for more details). These approaches introduced various metrics, such as Instructions Per Cycle (IPC) and Millions Instructions Per Second (MIPS), for determining the application workload on the processor through online and/or offline characterization. These metrics are used to take decisions on the selection of Voltage-frequency (V-f) setting.

For multi-threaded applications, the workload not only depends on memory-/compute-intensity but also on thread synchronization contention. In addition to that, some computing systems are usually based on Non-Uniform Memory Access (NUMA) architecture, such as AMD Opteron processor, Intel Itanium servers, etc., where memory access time depends on the memory location relative to the processor [160]. A thorough analysis of related works [109–111, 113–119], presented in Section 2.3.2, shows that the existing approaches do not consider the combined effect of the above factors while estimating the CPU workload. As a result, they are not efficient for adapting to workload variations, which is essential for improving energy efficiency. Furthermore, reported approaches do not take memory-contention into account when executing applications concurrently, resulting in increased power consumption without any performance benefits.

This work proposes a workload-aware runtime energy management technique that takes the aforementioned factors into account for improving the energy efficiency. Our proposal measures the processor workload using MRPI as a measure of memory-intensity and utilization for estimating the thread synchronization contention. For Intel based platforms, a metric called Memory Accesses Per Micro-operation (MAPM) is considered, which is equivalent to MRPI on ARM based platforms (please refer to Section 3.1.2.2). Moreover, latency due to NUMAs is calculated by monitoring the remote and local memory accesses during the application execution. As part of this, four hardware performance monitoring counters (PMCs) to compute MAPM, utilization and NUMA latency are used. To determine the appropriate V-f setting, a binning based approach is employed which takes utilization and MAPM as inputs. Furthermore, this approach works on both per-core and system-wide DVFS supporting platforms. Experimental validation is performed on the 12-core (24 threads) Intel Xeon E5-2630 and 61-core (244 threads) Xeon Phi many-core platforms. The former supports per-core DVFS, whereas the latter is based on system-wide DVFS. The main contributions of this work are:

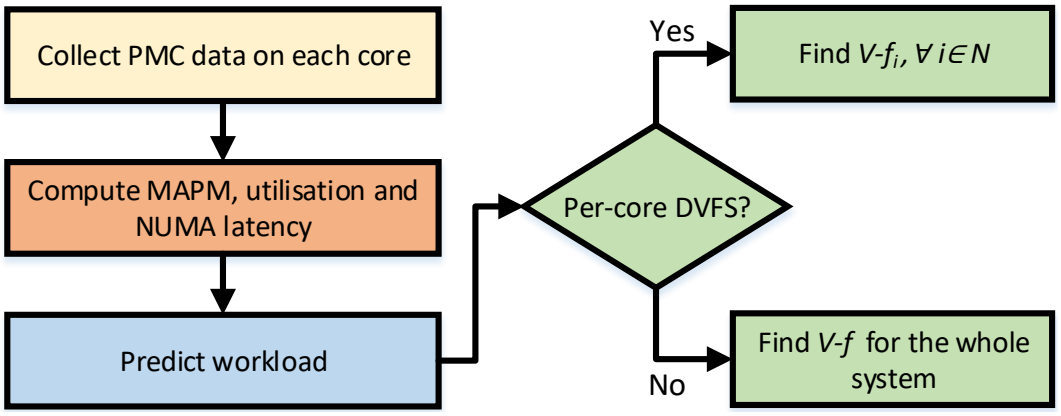

FIGURE 3.18: Illustration of various steps in the proposed approach for runtime energy management.

1. An accurate estimation of processor workload by considering the combined effect of memory-/compute-intensity, thread synchronization contention and NUMA latency;
2. A binning based approach for efficiently determining the *V-f* setting as per the predicted workload;
3. Validation of the proposed approach on two hardware platforms, the Xeon E5-2630 and Xeon Phi 7620P.

The reminder of the section is organized as follows. A detailed discussion of the problem formulation and the proposed approach is given in 3.3.1, while 3.3.2 presents an experimental validation of the proposed approach on two hardware platforms with various benchmark applications, including the runtime overheads before summarising the chapter.

3.3.1 Runtime DVFS Approach

The proposed approach considers two kinds of many-core platforms; one supporting per-core DVFS and the other one with system-wide DVFS. The following provides the problem definition, assuming that there are N cores in a platform.

Given a set of multi-threaded applications and a many-core platform supporting per-core/system-wide DVFS
Find an efficient *V-f* setting periodically for each core (V-f_i, where $i = 1, 2, ..., N$) or for the whole system such that the overall system energy consumption can be minimized while maximizing performance.

The proposed approach is illustrated in Fig. 3.18, which has the following four steps:

1. PMC data collection (Section 3.3.1.1);

2. Computing MAPM, utilization and NUMA latency (Section 3.3.1.2);
3. Workload prediction (Section 3.3.1.3);
4. Identification of *V-f* setting (Section 3.3.1.4).

The PMC data are collected periodically to get the information about the architectural events. The collected data is used to compute the MAPM, utilization and NUMA latency, which are further fed into a prediction algorithm for estimating the future workload. Considering the underlying architecture (per-core or system-wide DVFS), the *V-f* setting is determined according to the predicted workload. The detailed discussion on each step is given in the following sections.

3.3.1.1 PMC data collection

The modern processors support runtime monitoring of architectural events (e.g., instructions retired, cache misses, etc.) through a set of hardware performance monitoring counters (PMCs). These counters can be configured to count a particular architectural event from a list of supported events by a particular processor. The proposed approach needs `instructions retired`, `Last-Level Cache (LLC) misses` and `active CPU cycles`. It has been already shown in [53, 56] that similar events can be used to efficiently estimate the processor workload at runtime. Further, if the chosen platform has NUMA architecture, `remote memory accesses` are also collected. On the chosen platform (Xeon E5-2630 and Phi), the events matching to the above are the following symbolic names: `UOPS_RETIRED`, `LAST_LEVEL_CACHE_MISSES`, `UNHALTED_CORE_CYCLES`, and `MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM`. We use the tool `perfmon` [14] for periodically sampling the PMCs on each core simultaneously. This tool provides routines to configure the PMCs using symbolic names to count a particular event, to initialize, and to terminate the data collection.

3.3.1.2 Computing MAPM, utilization and NUMA latency

To find the appropriate frequency, the processor workload has to be determined accurately. This involves choosing a right metric and taking underlying memory architecture into account. Therefore, the metric Memory Accesses Per Micro-operation (MAPM) is chosen, similar to [56, 119], along with utilization and latency associated with non-uniform memory accesses. MAPM is computed as a ratio between memory accesses and micro-operations retired during the DVFS interval. This one has been used for measuring the memory-intensity of workload during application execution [56, 119]. Usually, a high value of MAPM suggests that the workload is memory-bound and vice versa. The memory accesses and micro-operations retired are measured using the PMCs, `LAST_LEVEL_CACHE_MISSES` and `UOPS_RETIRED`, respectively.

We have observed, especially in case of multi-threaded applications, the actual value of MAPM does not always represent the memory-intensity of the workload accurately. A lower value of MAPM may not always mean a compute-bound workload due to the following reasons. An application might have a lot of synchronization contentions, such as inter-thread locks and barriers or communication contentions, happening on external peripheral devices, e.g., storage devices, keyboard, etc. In such cases, MAPM alone fails to determine the actual load on the processor. To address this issue, along with MAPM, utilization is also considered for estimating the effect of aforementioned factors. To compute the processor utilisation, un-halted CPU cycles are used, which are monitored using the PMC `UNHALTED_CORE_CYCLES`.

Moreover, if a many-core platform is based on NUMA architecture, then the memory latencies differ depending on the relative distance between processor and memory. Access to the local memory is much faster than accessing the remote memory. Therefore, to efficiently calculate the effect of MAPM on processor performance, the NUMA latency should also be taken into account to select an appropriate *V-f* setting. To accomplish this, the number of remote memory accesses using the PMC `MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM` are measured for computing the MAPM. Assume that μ-*ops*, m_l and m_r represent micro-operations retired, accesses to local and remote memory, respectively. Then, MAPM can be determined as follows,

$$MAPM = \frac{m_l + \theta * m_r}{\mu\text{-}ops} \tag{3.7}$$

The constant θ depends on the latency associated with the remote memory access, which can be determined from the datasheet of the processor. As discussed in [56], the contention on memory, when applications are memory-bound, is also considered (refer to equation (3.8)).

3.3.1.3 Workload prediction

We use the workload predication, same as the one discussed in Section 3.2.1.3, for predicting the utilization and MAPM on each core to set *V-f* level proactively. The values of coefficients α and β are given in Section 3.3.2. Furthermore, as mentioned in Section 3.2.1.5, there is a possibility of having workload variations that are not significant enough to trigger switching of DVFS setting. Therefore, an adaptive sampling technique same as the one presented in the Section 3.2.1.5 is employed to minimize the runtime overheads when such scenarios are noticed.

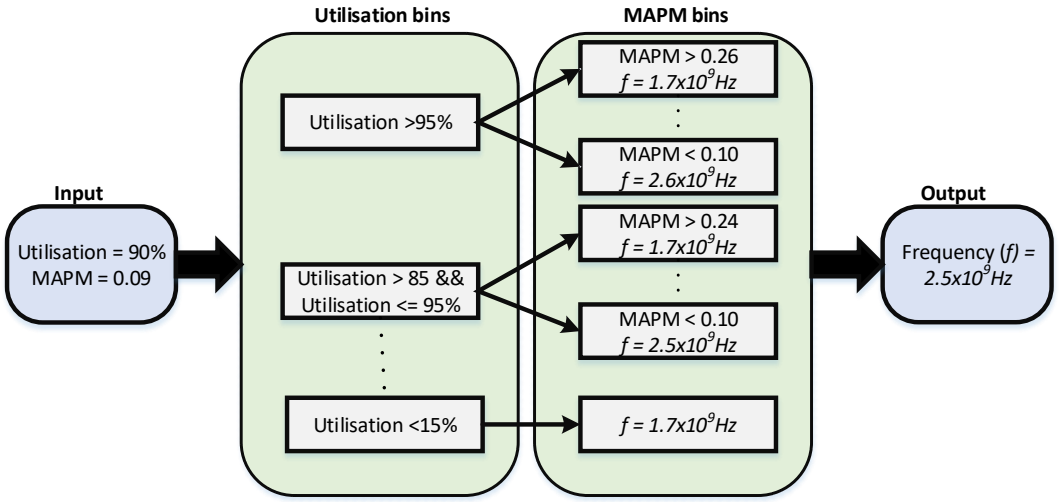

FIGURE 3.19: An example of *V-f* setting selection using binning based approach.

3.3.1.4 Identification of *V-f* setting

Determining the appropriate *V-f* setting is key to energy efficiency and to minimize performance loss. We employ a binning based approach [161] for finding the *V-f* setting based on the application workload. This approach consists of the two bins, one utilization bin and other one is MAPM bin. The utilization and MAPM computed from the step explained in Section 3.3.1.2 acts as inputs to this stage, as shown in Fig. 3.19.

Bins training: We use offline training to determine the bin boundaries. The first part of the training process is obtaining training samples. One training sample consists of the collection of metrics, MAPM and utilisation, along with the application performance at different *V-f* settings available on the chosen platform. In order to generate a diverse range of training samples, applications from different benchmark suites are considered, including SPEC CPU2006 [134], LMBench [136], RoyLongbottom [135], PARSEC 3.0 [15], NAS [17] and Rodinia [16]. These training samples are grouped into bins based on the *utilisation* and *MAPM*, and assigned a *V-f* setting to each bin that is energy efficient with no or little (<1%) performance loss. The experimental data used in classification bins is given in the Appendix B. During runtime, the utilization and MAPM are computed to find an appropriate *V-f* setting using the pre-determined bins. We use the utility `cpufreq-set` for changing the *V-f* setting during the application execution.

In case of per-core DVFS supporting platforms, the *V-f* setting of each core can be set as per its workload. But, if the platform supports only system-wide DVFS, the *V-f* setting of the whole system should be chosen in such a way that no application thread experiences the performance loss. Such cases can arise when applications with different workload types (e.g., compute-bound, memory-bound, etc.) are running concurrently. A memory-bound workload can be run at a lower *V-f* setting than a compute-bound,

but if both such workloads are executing concurrently, selection of an appropriate *V-f* setting is challenging. We use the similar approach proposed in [56] to address this issue by giving the preference to compute-bound workload over memory-bound one, and considering the impact of memory-contention due to concurrent execution. Therefore, the *V-f* setting is determined by the following MAPM value ($MAPM_t$),

$$MAPM_t = \min\{M_1, M_2, M_3, ..., M_N\} + \zeta * \sum_{i=1}^{N} \frac{M_i}{N} \tag{3.8}$$

Here, M_i, N and ζ represent MAPM of core i, number of processing cores in the system and coefficient to determine the effect of memory contention on the performance of each application thread, respectively. Furthermore, it can be understood that the result of minimum value computation in Equation (3.8) is same as the MAPM of the core with maximum utilisation because the most compute-intensive workload has minimum MAPM.

Identification of unused cores: MAPM of idle cores, executing no application thread, is usually low due to fewer memory accesses [53]. This gives a misimpression that such cores are executing a compute-intensive application, leading to the selection of a high *V-f* value for the whole system and thus increasing energy consumption. This becomes prominent when there are more cores than the number of concurrent applications. To address this, the proposed algorithm determines idle cores at runtime using utilization threshold. It is important to note that, for per-core DVFS supporting platforms, identification of idle cores is not required as the *V-f* setting is determined for each core, which is already taken care by the utilization bins, shown in Fig. 3.19. Based on experimental observation, if the utilization of a core is below 5%, it is identified as an idle core. Subsequently, MAPM of such cores is set to 10 (any value larger than one would be fine as the value of MAPM usually does not exceed one). This nullifies the influence of idle cores on *V-f* setting as it is mostly decided by the minimum MAPM (Equation (3.8)).

3.3.2 Experimental Results

The proposed approach is validated on an Intel Xeon E5-2630 running Red Hat Linux, and Xeon Phi coprocessor 7120P platforms. The platform details are given in Section 2.4.1, Chapter 2. The Rodinia [16] and NAS parallel [17] OpenMP benchmarks are used to demonstrate the efficacy of the proposed runtime energy management approach. These benchmarks are chosen, instead of the ones used in Section 3, because they offer execution time that are sufficiently large, and are mainly implemented for platforms like Xeon Phi. The classes of NAS and input to Rodinia benchmarks are chosen such that their execution times are sufficiently large (more than five seconds). The details of

TABLE 3.1: Selected applications from Rodinia [16] and NPB [17] and their abbreviations

Benchmark	Application Name	Abbreviation
Rodinia	Breadth-First Search	bfs
	HotSpot	hs
	K-means	km
	lavaMD	lmd
	Myocyte	mc
	Needleman-Wunsch	nw
	Particle Filter	pf
	Stream Cluster	sc
	Speckle Reducing Anisotropic Diffusion	srad
NAS	Block Tri-diagonal solver	bt
	Scalar Penta-diagonal solver	sp
	Lower-Upper Gauss-Seidel solver	lu
	Conjugate Gradient	cg
	Embarrassingly Parallel	ep
	Multi-Grid	mg
	Data Cube	dc

benchmarks are given in Section 2.4.2 and the selected applications and their abbreviations are shown in Table 3.1. For Xeon E5-2630 and Phi, the number of application threads is set to 24 and 61, respectively. These applications are executed in single, double and triple application scenarios. Similar to the experiments conducted in Section 3.2, the experimental results are collected by running each scenario ten times and finally, their average values (energy and performance) are computed for the comparison.

The proposed technique is compared against Linux's conservative (*CONS*), ondemand (*OD*) and performance (*PERF*) power governors, which are implemented on millions of devices, making them competitive baselines [155]. In addition to that, the approach presented in [119] is also considered for comparison. To demonstrate the advantage of taking the underlying NUMA architecture into account while estimating the workload, two variants of the proposed (*prop*) approach, *prop-NNUMA* (*NNUMA* stands for 'No NUMA') and *prop-NUMA*, are derived. As opposed to *prop-NNUMA*, *prop-NUMA* takes NUMA latency into account while deciding the *V-f* setting. For the better representation, the energy consumption of evaluated approaches is normalized to the energy consumption obtained by *prop-NUMA* (some applications have relatively lower energy consumption due to their shorter execution times).

Estimation of coefficients

The accuracy of the predicted workload as compared to the actual workload of the prior time intervals depends on the coefficients θ, γ and ζ (refer to Equation (3.7), (3.4) and

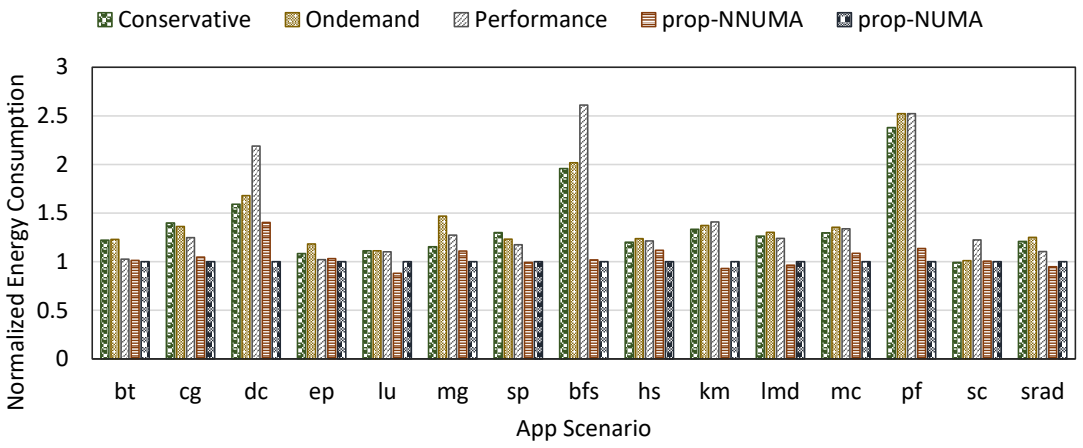

FIGURE 3.20: Comparison of the proposed technique with reported approaches for single application scenario in terms of normalized energy consumption, executing on the Xeon E5-2630.

(3.8)). The value of γ is computed from α and β (refer to Equation (3.5)). These coefficients are experimentally obtained by executing a diverse set of applications individually and concurrently. Finally, considering the relative workload prediction accuracy, θ, ζ, α and β are set to 2, 0.05, 0.3 and 0.6, respectively. The same coefficient values are used for all the application scenarios.

Evaluation of Workload Prediction

To evaluate the workload prediction accuracy, various application scenarios (executing single and multiple applications) are considered. For a set of 40 application scenarios, the average error in workload prediction was 5.4% with a minimum and maximum error of 0.5% and 9.3%, respectively.

3.3.2.1 Evaluation on 24-core Xeon E5-2630

Energy Savings

In the case of a single-application scenario, there is only one active application. The number of cores allocated to each application is 24; the same as the number of logical cores available on Xeon E5-2630. Fig. 3.20 shows a comparison of the adopted approaches, *prop-NUMA* and *prop-NNUMA*, with existing techniques in terms of normalized energy consumption. It can be observed that the proposed approach *prop-NUMA* outperforms existing approaches, except for the application *sc* executing under the *CONS* power governor. Moreover, *prop-NNUMA* also achieves better energy savings than reported approaches, except for applications *sc* and *ep* executing under *CONS* and *PERF* power governors, respectively. The proposed *prop-NUMA* achieves energy savings up to 57.9%,

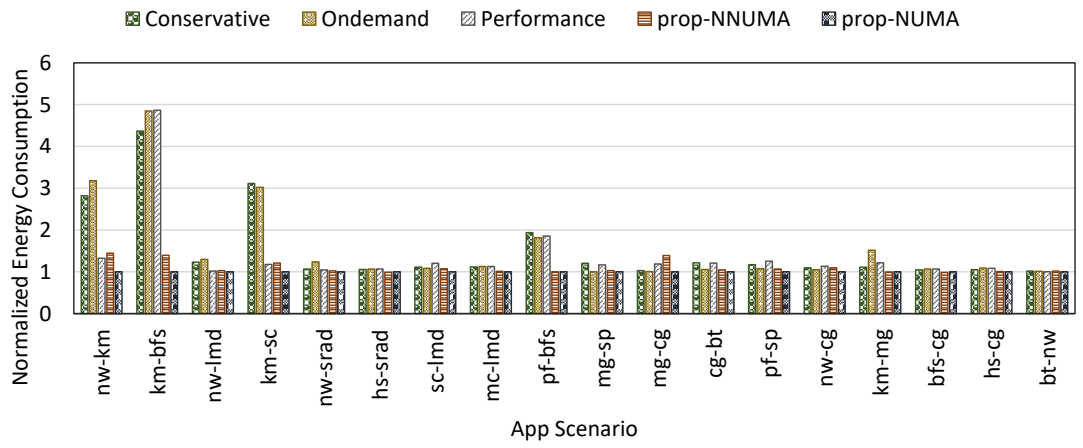

FIGURE 3.21: Normalized energy consumption of various approaches for double application scenario, executing on the Xeon E5-2630.

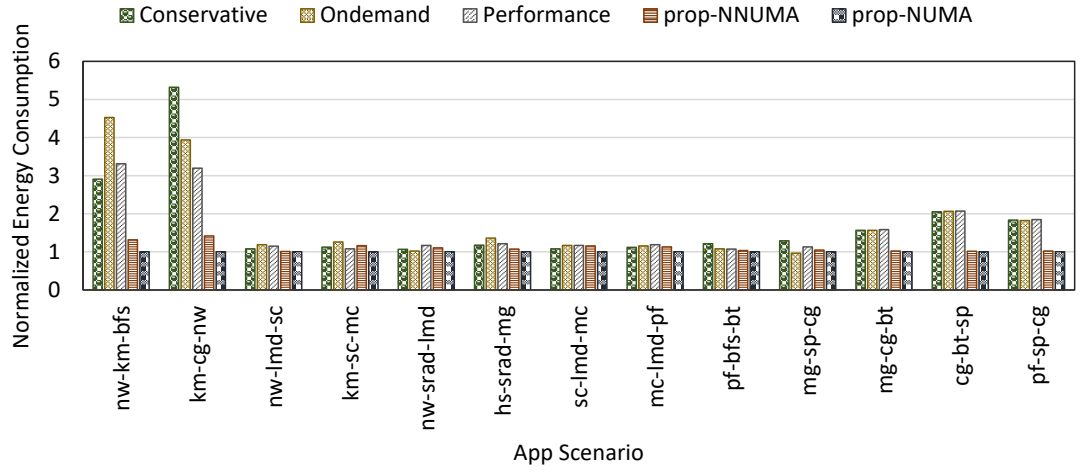

FIGURE 3.22: Comparison of the proposed approach with reported approaches for triple application scenario in terms of normalized energy consumption (evaluated on the Xeon E5-2630).

60.3%, 61.6% compared to *CONS*, *OD* and *PERF*, respectively. Furthermore, on an average, *prop-NUMA* consumes 3.2% less energy than *prop-NNUMA*.

For double and triple application scenarios, two and three applications are executed concurrently. In this evaluation, available cores on the platform are equally shared among the applications; for example, an application gets 12 cores in the double application scenario. This has been ensured by setting the core affinity of each application at the start of its execution. Concurrent execution increases the contention on memory, which actually suggests scaling down of *V-f* setting to exploit the increased data access latency for energy efficiency. Unlike existing approaches [119, 155], the proposed technique efficiently estimates memory contention, thereby minimizing energy consumption. Fig. 3.21 gives the normalized energy consumption for various approaches executing two applications concurrently. The proposed *prop-NUMA* approach improves energy consumption by up to 77.1%, 79.4% and 79.5% compared to *CONS*, *OD* and *PERF*, respectively. For

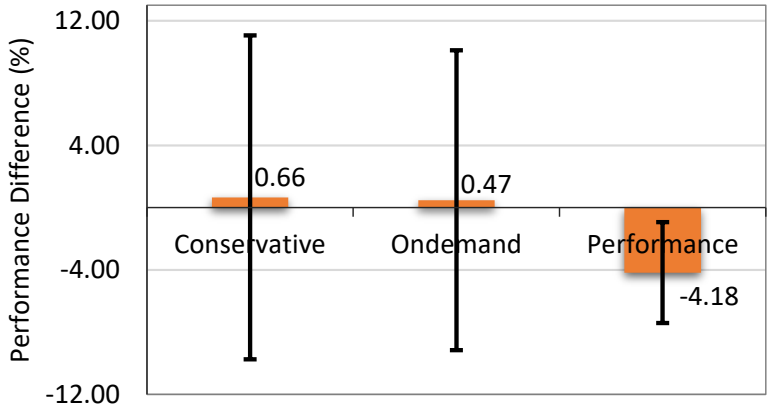

FIGURE 3.23: Mean performance difference between *prop-NUMA* and other approaches, with standard deviation error bars (evaluated on the Xeon E5-2630).

the application *bt-nw* scenario, *prop-NUMA* and *PERF* have similar energy consumption values. Furthermore, *prop-NNUMA* achieves an average energy saving of 7.8% with a standard deviation of 10.7% compared to *prop-NUMA*.

The proposed approach is also compared against the reported techniques in terms of normalized energy consumption for a triple application scenario. As shown in Fig. 3.22, *prop-NUMA* outperforms *CONS*, *OD* and *PERF* by up to 81.2%, 77.9% and 69.8%, respectively. From Fig. 3.20, 3.21 and 3.22, it can be observed that the average energy savings of *prop-NUMA* over *prop-NNUMA* keeps increasing, which is 9.4% with a standard deviation of 8.9%. This suggests that the advantage of considering NUMA latency is more evident when multiple applications are executing concurrently, leading to increased *LLC*-misses and remote memory accesses.

Application Performance

The application performance is evaluated for various application scenarios by computing the average execution time over 10 runs. The proposed technique achieves energy savings by scaling down the *V-f* setting if it does not result in performance loss. However, hardware platforms do not usually support fine-grained control of *V-f* setting, leading to performance degradation. Fig. 3.23 shows the mean and standard deviation of the performance difference (%) between *prop-NUMA* and other approaches. The average execution time for *prop-NUMA*, considering different application scenarios, is 0.66% and 0.47% less compared to *CONS* and *OD*, respectively. However, *PERF*, which runs at the maximum *V-f* setting, outperforms *prop-NUMA* by 4.18% (average). This shows that proposed approach improves energy efficiency with negligible performance loss in the most cases.

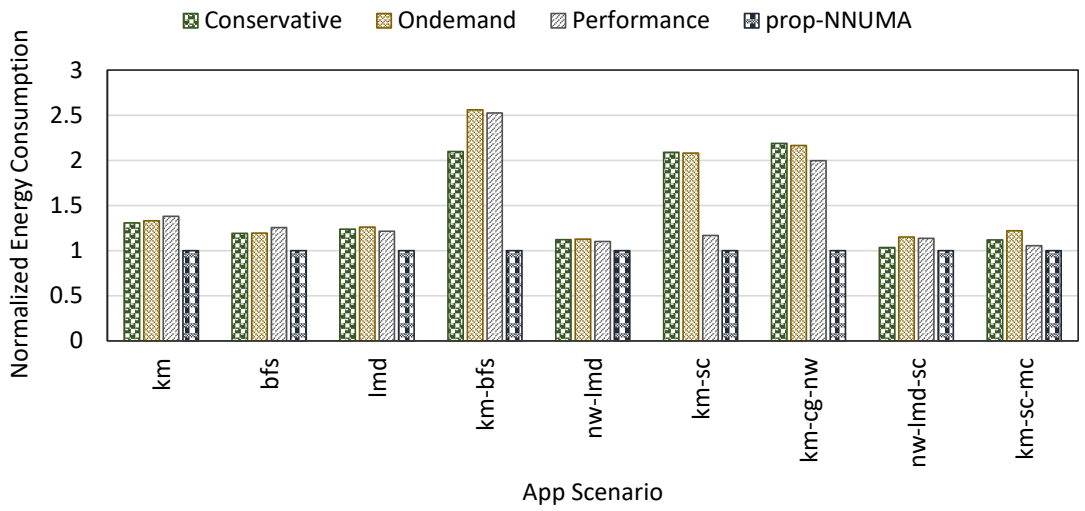

FIGURE 3.24: Comparison of proposed approach (*prop-NNUMA*) with reported approaches in terms of normalized energy consumption for various application scenarios, executing on the Xeon Phi.

3.3.2.2 Evaluation on 61-core Xeon Phi

Energy Savings

To show the effectiveness of the proposed approach on a platform supporting system-wide DVFS, experiments on the Xeon Phi with various application scenarios are also conducted. It should be noted that the Xeon Phi supports only simultaneous sampling of two PMCs. To address this issue, time multiplexing can be used to monitor more than two events; however, it increases the uncertainties in event count and runtime overheads [162]. Therefore, the `/proc` *sysfs* is used for measuring the core utilisation. Furthermore, the remote memory accesses, used in *prop-NUMA*, are not considered due to the above limitation. As shown in Fig. 3.24, on an average, considering all the application scenarios, *prop-NNUMA* improves energy efficiency by up to 54.3%, 60.9%, and 60.4% compared to *CONS*, *OD* and *PERF*, respectively. These energy savings relatively lesser (16%) compared to the ones achieved by per-core DVFS supporting platform (refer to Fig. 3.20, 3.21 and 3.22).

Application Performance

The mean and standard deviation of the performance difference between *prop-NNUMA* and other approaches are given in Fig. 3.25. Unlike the per-core DVFS platform, in this case the *V-f* setting of the whole system is decided by the most compute-intensive thread of all the executing applications. As a result, performance loss is minimized at the cost of lower energy efficiency, which is evident from the Fig. 3.24 and 3.25. The average execution time for *prop-NNUMA*, considering nine application scenarios, is

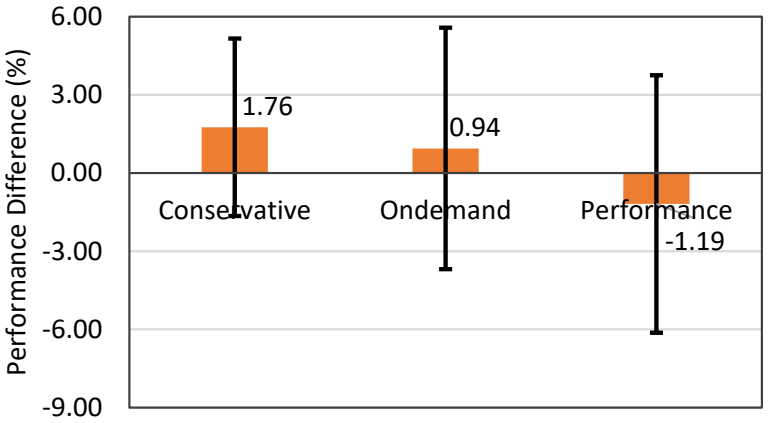

FIGURE 3.25: Mean performance difference between *prop-NUMA* and other approaches, with standard deviation error bars (evaluated on the Xeon Phi).

0.66% and 0.47% better compared to *CONS* and *OD*, respectively. But, as expected, *PERF* outperforms *prop-NNUMA* by 1.19%.

Comparison with Sundriyal *et al.* [119]

Sundriyal et al. [119] presented a model, which aims to predict the micro-operations retired at different frequencies and the MAPM values by using a linear regression analysis. It further chooses the scaling frequency at which the energy consumption is minimum by measuring the power using the Intel RAPL technology. This approach is proposed for single application scenario considering per-core DVFS. For comparison, the evaluation is carried out on the Xeon E5-2630 using *ep*, *cg*, *lu*, *mg*, *sp* and *bt* (same applications used in [119]). The approach proposed in [119] gives an average of 7.4% energy savings compared to *PERF* with a performance loss of 5.58%. Whereas, proposed approach (*prop-NUMA*) achieves an average energy savings of 17.8% with a performance loss of 1.8%. It shows that proposed approach outperforms the technique presented in [119] by 10.4% in energy efficiency and 3.7% in performance.

3.3.2.3 Runtime Overheads

Proposed approach involves sampling of four PMCs (only two on the Xeon Phi) on each core and subsequent processing to find and change the *V-f* setting. For all the application scenarios used in the evaluation, we have measured the runtime overhead of proposed approach on Xeon E5-2630 and Phi by monitoring the time spent in each decision epoch (200 ms). These steps mainly include, collection of PMC data, computation of MRPI and utilisation, prediction of workload, DVFS setting identification through classification bins and finally, adjusting the DVFS setting (*V-f* switching latency). The runtime overhead (T_o) is calculated with respect to the application execution time (T_{exec}) and this can be represented analytically as follows:

$$\%T_o = \frac{[T_{exec} - r \times T_p \times T_s] \times [T_{pmc} + T_{metrics} + T_{wp} + T_{classify} + T_{vfs}]}{T_s} \times 100 \quad (3.9)$$

where, r, T_p, T_s, T_{pmc}, $T_{metrics}$, T_{wp}, $T_{classify}$, T_{vfs} represent number of times the adaptation is paused; DVFS pausing length; DVFS sampling period; time required for collecting PMC data; MRPI/utilisation computation time; time taken by workload prediction; DVFS setting identification time (classification bins) and overheads associated with *V-f* switching (10 µs to 50 µs [163]); respectively. The experimental measurements shows a maximum overhead of 0.52% and 1.3% (in terms of application execution time) for Xeon E5-2630 and Phi platforms, respectively. The overheads have been measured by executing application to completion and logging the T_o when change in DVFS setting is noticed.

3.4 Discussion

This chapter has presented approaches for runtime management of DVFS through efficient workload classification. Firstly, an investigation on workload classification is carried out through statistical analysis of execution time and performance monitoring counter data. It has been observed that- (i) execution time of compute-intensive application scales proportionally with the frequency, whereas, for memory-intensive application, the change is minimal until the processing core frequency is higher than memory operating frequency, (ii) L2-cache read refills measure the memory-intensiveness of an application. Further, a metric, called Memory Reads Per Instruction (MRPI) for workload classification is derived to to efficiently make runtime decisions for adapting to application workload by tuning the DVFS setting.

As shown in the Section 3.2.2.4 and 3.3.2.3, the proposed runtime energy optimization technique, classifying the inherent workload characteristics of concurrently executing applications, incurs a lower overhead than existing learning-based approaches have significant overheads (up to 400 ms for a single application scenario [35]), which get further aggravated by dynamic workload variations causing frequent re-learning. Therefore, the scalability of such approaches in comparison to the proposed technique is limited for multi-core platforms executing multiple applications. The approach presented in Section 3.2 performs an on-the-fly workload classification using the metric MRPI and proactively selects an appropriate *V-f* setting for a predicted workload. Subsequently, it monitors the workload prediction error and performance loss, quantified by instructions per second (IPS), and adjusts the chosen *V-f* to compensate. Validation on a homogeneous multi-core platform (A15-cluster) for various combinations of applications shows an improvement of up to 69% energy efficiency compared to existing approaches.

Furthermore, the above approach has been extended to multi-threaded applications and per-core DVFS supporting many-core processor by incorporating thread-synchronisation and NUMA latency into workload estimation. A binning based approach with two classification layers is used, taking MAPM and utilisation as inputs, for identifying the appropriate DVFS setting. The proposed technique has been validated on two hardware platforms, Xeon E5-2630 and Phi, supporting per-core and system-wide DVFS. In the following chapter, this work is extended to a heterogeneous multi-core architecture running multiple multi-threaded applications by incorporating thread-to-core mapping and DVFS.

Chapter 4

Static Mapping and DVFS for Heterogeneous Multi-cores

The previous chapter showed that efficient online concurrent workload classification-based runtime management of DVFS results in better energy efficiency. However, the approaches presented in Chapter 3 do not consider heterogeneous multi-core platforms and thread-to-core mapping, which help improve energy efficiency. As shown in the literature review (Chapter 2), the heterogeneous multi-core architectures are computing alternatives for several application domains such as embedded [76] and cloud [164]. These architectures integrate several types of processing cores within a single chip. For example, ARM's big.LITTLE architecture contains two types of cores; big and LITTLE, where big cores are grouped into one cluster and LITTLE cores into another [165]. The big cluster has both higher cache capacity and computational power than the LITTLE one. In such architectures, distinct features of different types of cores can be exploited as per application characteristics and to meet end user requirements. These architectures are also equipped with dynamic voltage and frequency scaling (DVFS) to help save energy consumption. As discussed in the Chapter 3, these systems often deal with multiple applications concurrently while achieving the desired levels of performance for each of them. Moreover, applications have been implemented as multi-threaded to exploit multi-core chips, which can be mapped onto different cores for parallel execution, and thus reducing the overall completion time.

Efficient runtime management of multi-threaded applications on heterogeneous multi-cores is of paramount importance to achieve energy efficiency and high performance requirements, that have been a key research focus for computing systems [19, 34, 166]. In general, for each application, the management process first finds a thread-to-core mapping defining the number of used cores and their type, and then operating voltage-frequency levels of cores by looking the workload while satisfying the performance requirement. As part of these, the following experimental observations have been made.

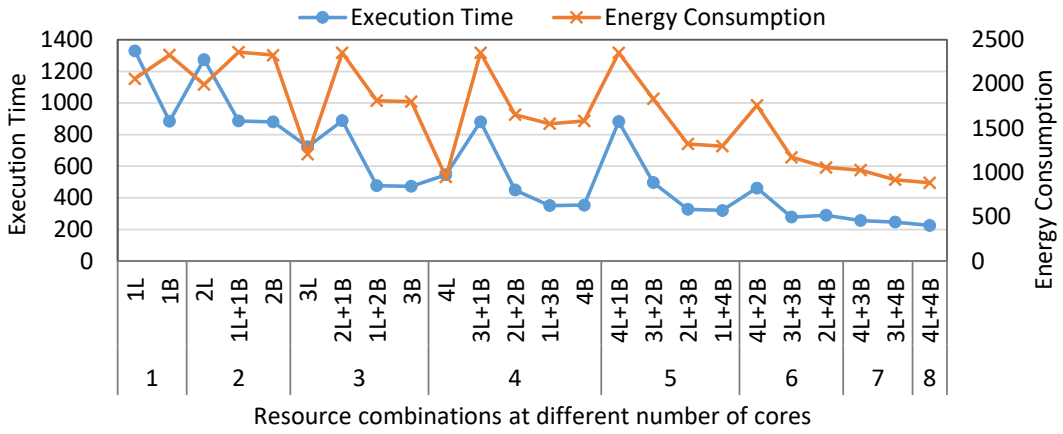

FIGURE 4.1: Execution time (seconds) and energy consumption (J) values by executing the Blackscholes application (from PARSEC benchmark [15]) with various core combinations, including inter-cluster, on ARM's big.LITTLE architecture containing 4 big (B) and 4 LITTLE (L) cores.

Observations: Fig. 4.1 shows the motivation to map an application on a heterogeneous multi-core architecture containing two types of cores, big (B) and LITTLE (L), where 4B and 4L cores are present. The horizontal axis shows various possible resource combinations. The vertical primary (left-hand side) and secondary (right-hand side) axes show execution time and energy consumption, respectively, when executing at the various resource combinations. The application execution time scales well with number of cores and further it benefits from mapping onto big and LITTLE clusters at the same time (referred to as inter-cluster thread-to-core mapping). It can be seen that executing on 4L and 4B cores is beneficial in terms of execution time and energy consumption, and thus thread-to-core mapping should utilize the 4B and 4L cores. Furthermore, as discussed in the Chapter 3, application workload can clearly be classified as compute-intensive, memory-intensive, and mixed (compute and memory-intensive), respectively, when multiple applications are run individually. However, it is completely different in the case of concurrent execution having greater workload variability due to applications' interference. Efficient workload classification in such scenarios is essential to chose appropriate DVFS levels to optimize energy while satisfying performance constraint.

The key steps in runtime management of concurrent execution of multiple applications on a heterogeneous multi-core system, are summarized in Fig. 4.2. Considering the above two observations made on inter-cluster exploitation and workload classification, following four main challenges are associated with the runtime management (described subsequently):

(i) Efficient inter-cluster thread-to-core mapping for multiple applications (left-most part of Fig. 4.2).

(ii) Workload classification for concurrently executing applications based on the identified thread-to-core mapping (middle part of Fig. 4.2).

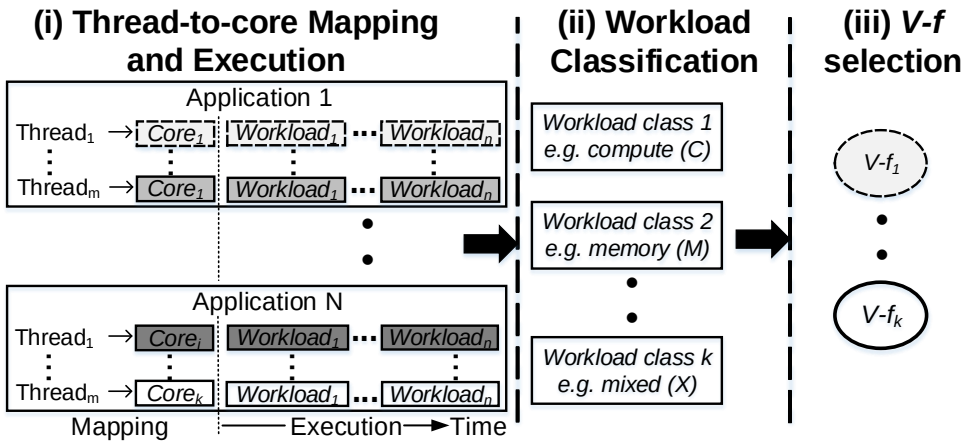

FIGURE 4.2: Key steps in runtime management of concurrent execution of multi-threaded applications on a heterogeneous multi-core architecture.

(iii) Identification of appropriate *V-f* level of cores for the associated workloads (right most part of Fig. 4.2).

(iv) Analysing concurrent applications' interference and taking appropriate measures to meet performance requirements.

(i) *Inter-cluster thread-to-core mapping* step needs to identify the number of used cores and their type for each application while meeting the performance requirement. In case of multiple applications, the mapping process has the challenge of allocating the right number and type of cores to each application from the available cores (4 LITTLE and 4 big cores for ARM's big.LITTLE architecture presented in the Odroid-XU3 [28]), such that their performance requirements are satisfied and energy consumption is minimized. The left-most part of Fig. 4.2 shows an example thread-to-core mapping for each application, where threads are allocated onto different types of cores (highlighted in various grey-scales).

(ii) *Workload classification* step needs to classify the workload within each cluster for the concurrently executing applications (shown under Execution in the left-most part of Fig. 4.2) by taking the workload of each core into account. The classification could be into various classes such as compute-intensive, memory-intensive and mixed [49,134], which results from the varying ways in which they exercise the hardware. From a performance perspective, it is desirable to run a compute-intensive application at a higher clock frequency as compared to memory-intensive one such that high performance can be achieved. However, an appropriate metric needs to be identified in order to classify the workloads, using MRPI in the present case. Classification at a given time interval plays a pivotal role for achieving the desired energy-performance trade-off, which is discussed in detail further in Section 4.2.

(iii) *Appropriate V-f identification* is required for the associated workloads such that the desired energy-performance trade-off can be achieved. Since multiple applications are

executed concurrently, the *V-f* level needs to be identified by taking the performance requirements of all of them into account. Further, as different set of *V-f* levels are available for the cores situated in different clusters, e.g., big and LITTLE clusters in ARM's big.LITTLE architecture, it becomes challenging to identify the most suitable *V-f* levels for different clusters while respecting applications' performance constraints. The right most part of Fig. 4.2 shows an example demonstration of *V-f* assignment.

(iv) *Applications' interference* due to concurrent execution of applications may degrade their performance. In order to meet the performance requirements, the interference level should be analysed and then it should be used to take appropriate measures. The interference level can be measured as the joint performance degradation of applications when executing concurrently in comparison to individual executions. Clustered heterogeneous architectures such as ARM's big.LITTLE represent different amounts of interference on big and LITTLE cluster for the same workload due to different amounts of available memory for them and thus they need to be analysed separately. Thereafter, they need to be exploited to meet the performance requirements.

Contributions: close observation of the existing runtime management approaches (described in the Section 2.3) indicates that they cannot address all the aforementioned challenges for executing multi-threaded applications on heterogeneous multi-core platforms. In order to overcome the limitations of the existing approaches towards addressing the above mentioned challenges, this chapter makes the following contributions:

1. Offline analysis of individual applications for performance and energy consumption when mapped to various possible resource combinations on a given heterogeneous multi-core platform to obtain profiled data.
2. For concurrently executing applications, an online mapping strategy facilitated by sorted profile data, to compute the minimum energy consumption point, while satisfying the performance and resource constraints. For each application, the computed point defines thread-to-core mapping, and the platform is configured following the mapping to start the application execution.
3. For the chosen thread-to-core mappings of concurrently executing applications, an online energy optimization technique that extends the DVFS technique presented in Section 3.2 to heterogeneous multi-cores.
4. Implementation and validation of both the offline and online steps on a hardware platform, the Odroid-XU3 [28].

The remainder of the chapter is organized as follows. Section 4.1 discusses problem definition in detail, while Section 4.2 describes various stages of the proposed methodology. Section 4.3 presents the experimental results and their analysis with chosen benchmark applications and hardware platform. Finally, Section 4.4 ends the chapter with a summary.

4.1 Problem Formulation

For an application with R threads to be mapped onto a heterogeneous multi-core architecture with N clusters, i.e. N core types (C_1, C_2, C_3, ..., C_N), where each cluster contains l_i ($i = 1, ..., N$) cores, the possible number of thread-to-core mappings (TC_{map}) can be represented as,

$$TC_{map} = \begin{cases} \sum_{i=1}^{N} l_i + \prod_{i=1}^{N} l_i & R \geq \sum_{i=1}^{N} l_i \ \& \ R \geq l_i \\ R * N + R^N & R < \sum_{i=1}^{N} l_i \ \& \ R < l_i \end{cases} \tag{4.1}$$

For runtime power management, the modern cluster-based architectures support cluster-wide DVFS, where cores of the same type organized as a cluster are set to the same *V-f* level from a predefined set of *V-f* pairs [165]. For example, Odroid-XU3 [28], Juno r2 [29], and Mediatek X20 [30] platforms employ such architecture. Let l_i be the number of cores of type C_i in a cluster E_i and nF_i be the number of available *V-f* levels. Then, to incorporate the *V-f* levels (nF_i) into thread-to-core mapping decisions, equation 4.1 is modified as follows,

$$TC_{map\_VF} = \begin{cases} \sum_{i=1}^{N} l_i * nF_i + \prod_{i=1}^{N} l_i * nF_i & R \geq \sum_{i=1}^{N} l_i \ \& \ R \geq l_i \\ \sum_{i=1}^{N} R * nF_i + \prod_{i=1}^{N} R * nF_i & R < \sum_{i=1}^{N} l_i \ \& \ R < l_i \end{cases} \tag{4.2}$$

As can be seen from equation 4.2, the initial design space is prohibitively large to explore during the application execution at different time intervals, and thus cannot be applied at runtime. In order to overcome this issue, the exploration of mapping can be fixed to the initial design space, and DVFS exploration can be carried at runtime during different time intervals by fixing the mapping from the initial design space. This helps to solve the thread-to-core mapping and DVFS problems orthogonally. The problem has been tackled in the same manner, as defined below:

Given an active application or a set of active applications with performance constraints and a clustered heterogeneous multi-core architecture supporting DVFS

Find an efficient static thread-to-core mapping for each application at runtime and then apply DVFS during the application execution to minimize the energy consumption

Subject to meeting performance requirement of each application without violating the resource constraints (number of available cores in a platform)

For a total of n applications, there are 2^n possible use-cases, where each use-case represent a set of active applications. Finding all the possible mappings for each use-case

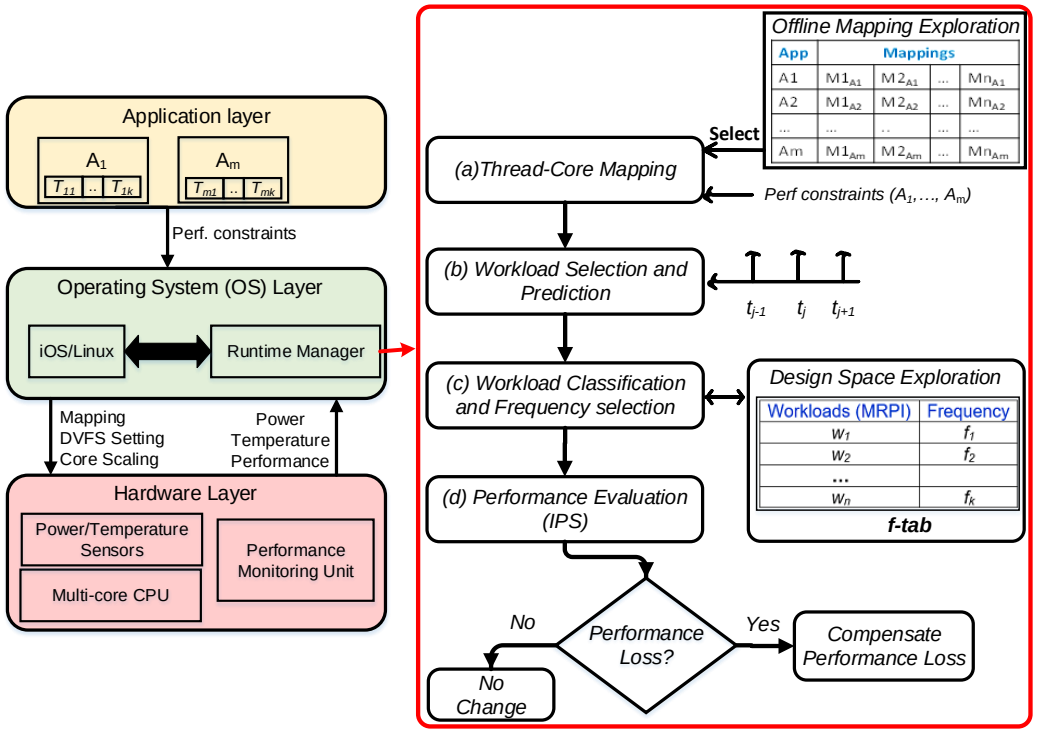

FIGURE 4.3: Overview of a three-layer representation of a multi-core system (left) and proposed runtime manager (right).

might not be possible within a limited time in case the number of applications and/or cores in each heterogeneous cluster increases. Therefore, the mappings can be explored for an individual application and used in conjunction for various use-cases at runtime, which also reduces the overhead to store the mappings. The same measures are employed in the proposed approach.

4.2 Static Thread-to-Core Mapping and DVFS Approach

A three-layer view of a typical multi-core system is presented in Fig. 4.3 (left), where each layer interacts with the others to execute an application, as indicated by arrows. The top most layer is the application layer, which is composed of multiple applications having various workload classes. The middle layer is the operating system layer (e.g. iOS, Linux, etc.), which coordinates an application's execution on the hardware (bottom), consisting of multi-core processors. An overview of the proposed runtime management approach employed by the OS has been illustrated in Fig. 4.3 (right), which has the following stages:

(a) Thread-to-core mapping (Section 4.2.1)

(b) Workload selection and prediction (Section 4.2.2.1)

(c) Workload classification and frequency selection (Section 4.2.2.2)

(d) Performance observation and compensation (Section 4.2.2.3)

A detailed discussion of each stage is presented in the following sections.

4.2.1 Thread-to-Core Mapping

In order to meet the performance requirements of the applications to be run concurrently, and to minimize the energy consumption, an effective thread-to-core mapping is important. This involves choosing an appropriate number of cores and their type for each application. Since there are several thread-to-core mapping options for each application, exploring the whole mapping space is time consuming. Therefore, at runtime, thread-to-core mapping is facilitated through an extensive offline analysis to guide application execution towards an energy efficient point. The offline analysis evaluates performance, and energy consumption for all the possible thread-to-core mappings at the maximum available frequency that helps to meet high performance requirements. At runtime, one of these mappings that leads to energy-efficiency while meeting performance and resource constraints is selected for each application. The following sections present a detailed discussion on offline analysis and runtime mapping selection.

4.2.1.1 Offline Analysis

For each available application, the offline analysis computes all the possible thread-to-core mappings and their performance and energy consumption on a given cluster-based heterogeneous architecture. For the considered applications, the number of threads is greater than the number of cores available on the chosen hardware platform (Odroid-XU3). Therefore, following equation 4.1, the total number of thread-to-core mappings for each application is 24 (TC_{map} =4+4+4*4). Fig. 4.1 presents an example analysis for the Blackscholes and Bodytrack applications that shows 24 mappings and their respective performance (1/execution time) and energy consumption.

The analysis results for each application are stored as design points that represent performance and energy trade-off points for all possible thread-to-core mappings at the maximum frequency. Each design point is represented as 4-tuple: (Prf, EC, n_L, n_b), where Prf, EC, n_L, and n_b denote performance, energy consumption, number of LITTLE cores, and number of big cores, respectively. These design points are sorted in descending order to quickly identify the points meeting a certain level of performance. This helps in minimizing the runtime overhead while choosing the minimum energy point for each performance-constrained application. Fig. 4.4 shows the design points for the *Blackscholes* application corresponding to Fig. 4.1. Similarly, design points are

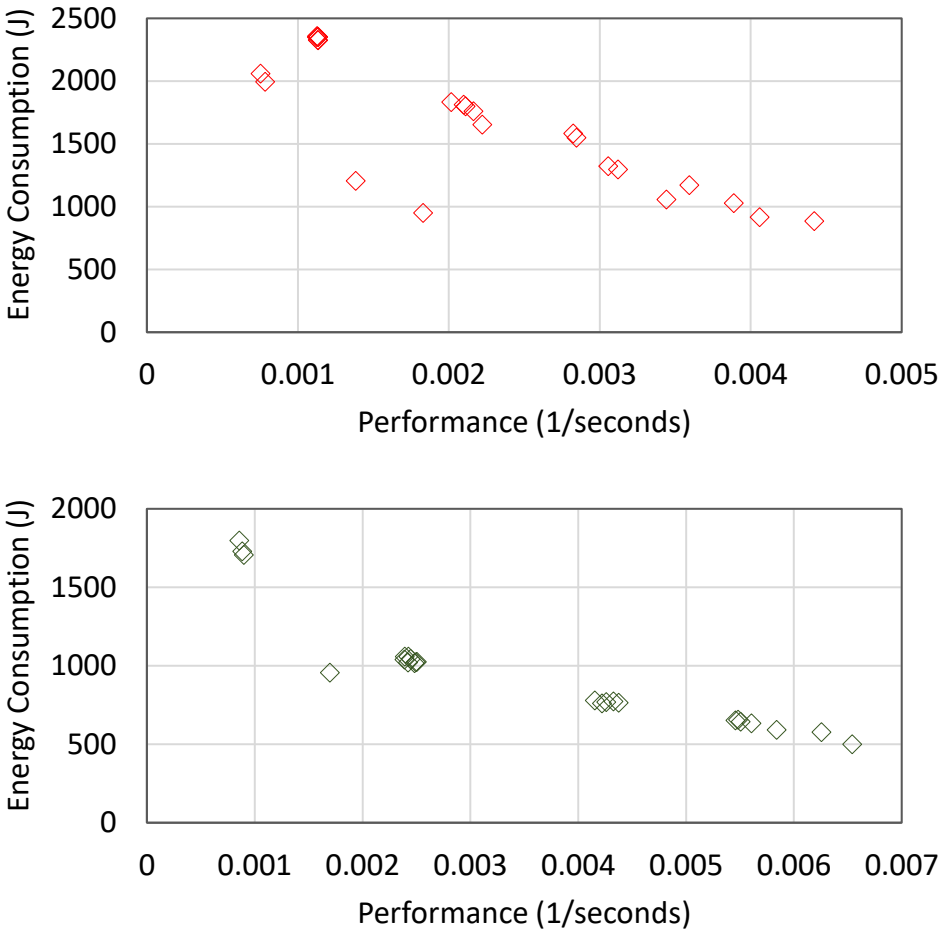

FIGURE 4.4: Design points representing performance and energy trade-off points for `Blackscholes` (top) and `Bodytrack` (bottom) applications.

stored for other applications as an outcome of the offline analysis. The application offline analysis results are presented in the Appendix A.

4.2.1.2 Runtime Mapping Selection

For a set of active applications, the runtime mapping has to identify appropriate design points for each application such that the overall energy consumption is minimized without violating the performance and resource constraints (number of cores available in the platform). For example, in case of the Odroid-XU3 platform, the total number of used big and LITTLE cores should not exceed four of each.

Algorithm 2 describes the runtime mapping selection algorithm. The algorithm takes concurrently active applications (CA_{Apps}), their performance requirements ($Apps_{Prfr}$) and design points (generated in the previous step, $DP = D_1, ..., D_m$) as input and provides a static thread-to-core mapping Map in terms of number of used cores and their types as output for each application. It has been observed in [40] that allocating

Algorithm 2 Runtime thread-to-core mapping selection

Input: CA_{Apps}, $Apps_{Prfr}$, DP
Output: Map for each application
1: **for** each application A_m **do**
2: Choose points D_{A_m} ($\in DP$) such that $Prf > Am_{Prfr}$;
3: **end for**
4: **for** each combination point CP (from CA_{Apps}) **do**
5: Compute energy consumption of CP as $Energy[CP]$ (Equation 4.3);
6: Compute total number of used cores of different types (Equation 4.4);
7: Add CP with its $Energy[CP]$ and $C_{i\_UsedCores}[CP]$ in set CPS;
8: **end for**
9: From CPS, select the point having minimum energy consumption ($minEnergy[CP]$) and satisfying resource constraint (i.e., $C_{i\_UsedCores}[CP] < available\ C_{i\_Cores}$);
10: For the $minEnergy[CP]$, return number of used cores and their types for each application as Map;

more number of threads than cores does not actually give any performance benefits. Moreover, by varying number of threads per core (1, 2, .., t) where the value of t can vary depending on the application and resource allocation), the design space becomes prohibitively large. Therefore, to reduce the mapping complexity, the number of threads are chosen the same as the number of cores. For each application, the algorithm first chooses performance requirement satisfying points from its design points. Since the points are stored in decreasing order of performance, the points are chosen as the first entry in the storage to the last entry meeting the performance requirement. Thus, a quick selection of points take place. Then, for each combination point CP (formed by considering one point from each application), energy consumption and used cores of type C_i are computed as follows.

$$Energy[CP] = \sum_{m=1}^{NrCApps} Energy_m \tag{4.3}$$

$$C_{i\_UsedCores}[CP] = \sum_{m=1}^{NrCApps} C_{i\_cores_m} \tag{4.4}$$

where, $Energy_m$ and $C_{i\_cores_m}$ are the energy consumption and used C_i type cores of application m, respectively. For $NrCApps$ active applications, a combination point contains one point from each application.

After above computations for different combination points, the point having minimum energy consumption (based on minimum value selection algorithm) and satisfying the resource constraint is chosen (line 9, Algorithm 2). Then, for this chosen point, the

number of used cores and their types (n_L and n_b) for each application are returned as the thread-to-core mapping Map, which is further controlled by `sched_setaffinity` interface in the Linux scheduler. Our approach is generic, but one-time offline analysis is required when the application or platform changes. In case a new application needs to be executed and its offline analysis is not done, the best effort or online learning heuristics [50] can be employed to obtain the mapping, but achieved results might not be efficient.

Simultaneous execution of multiple applications may affect each other due to interference in the shared memory. Thus, their execution time might increase in comparison to the scenario when executed individually. The stretching in the execution time depends upon the compute and memory-intensiveness of the applications, e.g. higher stretching is expected for memory-intensive applications due to heavy memory accesses in the shared memory. The degraded performances of the applications are compensated by proper *V-f* selections during various time intervals during execution, which is described in the following sub sections.

4.2.2 DVFS Approach

The DVFS approach presented for homogeneous multi-cores in the Chapter 3 has been extended to heterogeneous multi-cores supporting cluster-based DVFS. The pseudo code of the proposed online DVFS is given in Algorithm 3.

4.2.2.1 Workload Selection and Prediction

In a cluster based architecture, the *V-f* of an individual cluster should be decided by considering the workloads of all the cores within a cluster (W_{E_i}) which can be represented as:

$$V\text{-}f_{E_i} = f(W_{E_i}) \tag{4.5}$$

It is important to note that, for concurrent execution of multi-threaded applications, the *V-f* setting of each cluster should be chosen in such a way that all the applications meet their performance requirements. Furthermore, these applications generate varying and mixed workloads due to resource sharing (e.g. L2 cache and memory) showing intra and inter workload variations during their execution. Therefore, a representative *V-f* pair needs to be selected to guide the further stages in achieving energy efficiency without performance loss. Pseudo code of the proposed online DVFS is given in Algorithm 3.

Assume that there are R concurrently executing threads of application(s) on cluster E_i, and w_{rj} is the workload of a thread r for time interval $t_{j-1} \rightarrow t_j$. There will be R

Algorithm 3 Adaptive DVFS approach

Input: Application scenario, T_s, *f-tab* and *len*
Output: *V-f* pair for each cluster

Initialisation: predicted workload (P_w)=0, c_l=c_b=0;
PMUINITIALIZE()
f_{cur} = `cpufreq_get_frequency (core#)`
while (1) **do**
    compute new IPS value (IPS_n) for each application
    wait for T_s /*DVFS interval*/
    /*Set V-f level of A15-cluster*/
    if ($*c_b \neq len$) **then**
        actual workload(A_w) = $Pmc\_data\_A15()$
        prediction error (P_e) = $A_w - P_w$
        $q = 0$
        FIND_SET_VF($A_w, P_w, P_e, c_b, core\#$)
        $q = 1$
    else
        wait for $T_s * len$ /*Adaptive sampling*/
        $*c_b = 0$
    end if
    /*Set V-f level of A7-cluster*/
    if ($*c_l \neq len$) **then**
        actual workload (A_w) = $Pmc\_data\_A7()$
        prediction error (P_e) = $A_w - P_w$
        FIND_SET_VF($A_w, P_w, P_e, c_l, core\#$)
    else
        wait for $T_s * len$ /*Adaptive sampling*/
        $*c_l = 0$
    end if
end while
function FIND_SET_VF($A_w, P_w, P_e, c, core\#$)
    P_w = EWMA($P_w, A_w, P_e, \&c$)
    classify P_w, compute δ and get f_n from $f - tab$
    if $q == 0$ **then**
        for `each application` **do**
            $Perf\_loss = ((IPS_{req} - IPS_n)/IPS_{req})$
        end for
        $\lambda = Perf\_loss * 100$
    end if
    if ($\lambda > x\%$) **then**
        $f_n = f_n + \lambda * f_{max}$
    end if
    if ($f_{new} \neq f_{cur}$) **then**
        `cpufreq_set_frequency (core#,f`$_n$`)`
        f_{cur}=f_{new}
        *c−−
    else
        *c++
    end if
end function
PMUTERMINATE()

TABLE 4.1: Design time analysis of workload classes (MRPI range) and corresponding frequencies

A15		**A7**	
MRPI Range	Frequency (MHz)	MRPI Range	Frequency (MHz)
>0.036	1000	>0.055	900
(0.033,0.036]	1100	(0.05,0.055]	1000
(0.03,0.033]	1200	(0.044,0.05]	1100
(0.027,0.03]	1300	(0.04,0.044]	1200
(0.024,0.027]	1400	(0.032,0.04]	1300
(0.021,0.024]	1500	<0.025	1400
(0.018,0.021]	1600		
(0.015,0.018]	1700		
(0.012,0.015]	1800		
(0.009,0.012]	1900		
<0.009	2000		

different workloads at every time interval of execution. The workload is quantified by the MRPI, where a low value represents a high load on the core and vice versa. If there are multiple workloads running within a cluster, choosing a *V-f* setting based on an average or single workload may lead to violation of performance requirements for some of the applications. Therefore, as discussed in Section 3.2.1.2, workload that decides DVFS setting for cluster E_i for the time interval $t_{j-1} \rightarrow t_j$ is computed as follows (line 9 and 20 in Algorithm 3):

$$W_{E_{ji}} = min\ (w_{1ji}, w_{2ji}, w_{3ji}, w_{4ji}, ..., w_{Rji}) + \delta_{ji} \tag{4.6}$$

The δ, signifying the effect of memory contention on application execution time, is computed from average MRPI of the cores within a cluster (please refer to Section 3.2.1.2 for more details). Experimentally analysis shows that δ increases the application execution time from 1.08% to 3.80%, when multiple applications are executed concurrently. Based on the above observation, the δ value is set to 4.5% of average MRPI.

Proactive control of *V-f* is of utmost importance for online energy. Therefore, the future workload needs to be predicted at t_j to set the *V-f* pair for the interval $t_j \rightarrow t_{j+1}$. The same workload prediction technique presented in Section 3.2.1.3 is used (line 29, Algorithm 3) and its corresponding experimental evaluation is given in Section 4.3.

4.2.2.2 Workload Classification and Frequency Selection

It is essential to classify the predicted workload for identifying an appropriate *V-f* pair for meeting the performance requirements and optimizing the energy. We use hardware PMCs for periodically getting information regarding architectural parameters during application execution (line 9 and 29, Algorithm 3). The modified performance monitoring tool `perfmon` [149] (enabled access to the A15- and A7-clusters) is used for accessing the PMCs, initialized and terminated through `PMUINITIALIZE` and `PMUTERMINATE` (line 2 and 48, Algorithm 3). Section 3.2.1.4 of Chapter 3 already presented a methodology for identify the appropriate *V-f* setting for different workload classes on A15-clusters. The same methodology has been used for A7-cluster.

The A15-core is an out of order core which takes advantage of memory level parallelism such that part of an L2 cache miss latency overlaps with other independent L2 cache misses. Furthermore, the A15-cluster has greater L2-cache capacity compared to the A7-cluster. The influence of these factors is seen during exploration, and taken into account while choosing the MRPI range and its corresponding frequency. The classified workloads and corresponding optimal *V-f* settings obtained from *f-tab* are given in Table 4.1, where various MRPI ranges are mapped to frequency through DSE, and used at runtime to set the operating frequency to a desired value through the utility `cpufreq-set`, thereby minimizing runtime overheads (lines 41, Algorithm 3).

4.2.2.3 Performance Observation and Compensation

Each application, especially in a concurrent execution scenario, may experience performance degradation due to variations in the application workload and interference from co-scheduled applications. Therefore, the technique presented in Section 3.2.1.6 is used for measuring each application performance and compensate it when a performance requirement is violated. The performance loss is calculated once (lines 31-35, Algorithm 3) by comparing the achieved IPS (IPS_n) of each application with their required IPS (IPS_{req}) at every time interval. More details are given in Section 3.2.1.6.

Adaptive Sampling

As discussed in Chapter 3, sometimes workload variations during execution may not be significant enough to cause a change in DVFS setting. Therefore, the approach proposed in Section 3.2.1.5 is extended to incorporate both A7- and A15-clusters to pause DVFS algorithm when there are unnoticeable workload variations. This achieve this, counters c_b and c_l are used for tracking the workload variations on A15- and A7-clusters, respectively. These counters get incremented when the workload at t_j is significantly different (MRPI range) than that of the workload at t_{j-1} (lines 40-46, Algorithm 3).

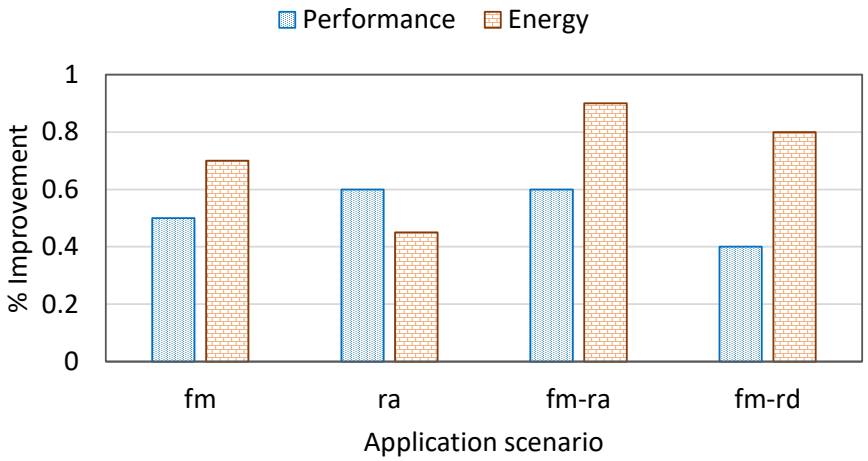

FIGURE 4.5: Effect of adaptive sampling on energy and performance for various application scenarios.

TABLE 4.2: Selected applications from PARSEC [15] and SPLASH [18] benchmarks

Benchmark	App Name	Abbreviation
PARSEC	blackscholes	bl
	bodytrack	bo
	swaptions	sw
	freqmine	fr
	vips	vi
SPLASH	water-spatial	wa
	raytrace	ra
	fmm	fm

When c_b or c_l is equal to a configurable parameter *len*, the runtime adaptation on A15- or A7-cluster (PMC data collection and subsequent processing) is paused for len^*T_s period (lines 15 and 24, Algorithm 3). We evaluated the effect of adaptive sampling for the application workload scenarios: fm, ra, fm-ra and fm-rd, where an average improvement of 0.9% and 0.6% in energy and performance are observed respectively when adaptive sampling is enabled, as shown in Fig. 4.5.

4.3 Experimental Validation

The proposed runtime management approach for energy optimization is extensively validated on an Odroid-XU3 platform running Ubuntu Linux Kernel 3.10.96 with a number of combinations of applications from PARSEC [15] and SPLASH [18] benchmarks. The selection of these benchmarks is mainly motivated by the availability of a diverse set of workloads (exhibit different memory behavior, data partitions and data sharing patterns) that can be used evaluate concurrency and parallel processing. Moreover, they support changing the number of threads for each application without recompilation. The

TABLE 4.3: Approaches considered for comparison

Reference	Approach	Abbreviation
[50]	Exhaustive Search-based	ES
[10, 125–127, 168]	Workload Memory Intensity based thread-to-core mapping	WMI
[169, 170]	HMP + Ondemand	HMPO
[169, 170]	HMP + Performance	HMPP
[169, 170]	HMP + Conservative	HMPC
[169, 170]	HMP + Interactive	HMPI
proposed	Inter-cluster Thread-to-core Mapping and DVFS	ITMD

details of the Odroid-XU3 platform are already provided in Section 2.4.1 of Chapter 2. We selected applications from PARSEC and SPLASH, based on variations in MRPI values. Table 4.2 lists the considered applications. The applications are taken in various combinations to make sets of simultaneously active applications and their abbreviations. To have better predictability and to ensure that each application meets its performance requirement, the system is not overloaded, i.e. no two applications share the cores. This allows scheduler not to delay the application execution. If applications arrive at different times, the later arrived ones can be mapped by taking the available resources and current status of the existing applications, computed as the remaining time to complete them. If existing applications are going to complete soon, the freed resources by them can be considered to decide the mapping of the current application, otherwise it should be decided based on the current available resources. This also avoids the overhead of data transfer for existing applications as their mapping is not disturbed.

The presented approach is implemented as a *user space* application by using the `Perfmon2` [167] and `cpufrequtils` framework (more details are given in Section 2.4.3. The `Perfmon2` tool enables the *user space* access to the performance monitoring unit (PMU), and `cpufrequtils` helps in setting/getting the operating frequencies. To bind an application to a core or a set of cores, the Standard Linux API `sched_setaffinity(2)` is used which controls the CPU affinity of processes. Energy consumption is calculated as a product of average power consumption (dynamic and static) and execution time. This includes both the core and memory energy consumption of all the software components (proposed algorithms (Algorithm 2 and 3), profiled data, OS, applications, etc.). Similar to previous experiments presented in earlier chapter, the results are collected by running each scenario ten times and finally, their average values (energy and performance) are computed for the comparison.

The proposed runtime management approach is compared against various approaches, given in Table 4.3, to show energy savings while satisfying the performance constraints. As part of these, the state-of-the-art solution for the runtime resource management of the

big.LITTLE, Heterogeneous Multi-Processing (HMP) scheduler [169] with various DVFS governors (ondemand, performance, conservative and interactive) is considered. HMP is a patch to the standard scheduler in the Linux kernel which dynamically dispatches threads to big and LITTLE cluster according to their characteristics.

Furthermore, a mapping approach, which allocates the application's threads onto only one type of core(s) based on memory-intensiveness [10, 125–127, 168] is considered for the comparison. As concurrent execution of multi-threaded applications is not taken into account by the above approaches, following changes are made for a fair comparison.

- In case of single-application scenario, a memory-intensive application threads are mapped onto LITTLE cluster.
- In multiple-application scenario, applications are sorted based on their memory-intensiveness and then one with the high memory intensity is mapped onto little cores and remaining applications are allocated onto big cluster with equal number of cores.

The proposed approach is also compared against a recently published exhaustive search-based (ES) approach [50]. As part of this, the thread-to-core mappings produced by proposed approach are used and varied the frequencies (247 design points; 200 MHz - 1400 MHz on LITTLE-cluster and 200 MHz - 2000 MHz on big-cluster) for different application scenarios. Then, selected the configuration (number of cores and their frequencies), having the lowest energy consumption while satisfying the performance requirements. Dimitrios et al. [171] presented a process variation- and workload-aware thread-to-core mapping approach on heterogeneous multi-core systems. However, this approach was not considered for the direct comparison with the proposed approach for the following reasons. It is proposed for maximizing the throughput under both performance and power constraints, while proposed approach minimizes the energy consumption under performance constraints. Moreover, the system architecture is different than the one (cluster-based architecture) used in this work.

To show the effectiveness of the proposed methodology compared to various existing approaches in terms of energy savings and performance, single and multiple-application scenarios are considered for the validation. Moreover, the validation of the workload prediction is also presented.

4.3.1 Energy Savings and Performance Comparison

4.3.1.1 Energy Savings

Table 4.4 presents the resource combinations achieved by the ITMD's mapping approach at runtime for various application scenarios. The mapping approach takes the individual

TABLE 4.4: Resource combination achieved by proposed mapping approach at runtime for different application scenarios.

App scenario	App combination	Resource combination
single	bl	4L+4B
	bo	4L+4B
	sw	4L+4B
	fr	4L
	wa	4L+4B
	ra	3L+4B
double	bl-bo	2L+2B : 2L+2B
	bl-sw	4B : 4L
	fr-sw	4L : 4B
	wa-bo	2L+2B : 2L+2B
	wa-bo	4L+3B : 1B
	wa-ra	4L+3B : 1B
triple	bl-bo-sw	3B : 1B : 4L
	bl-bo-fr	3B : 1B : 4L
	sw-bo-fr	4L : 1B : 3B
	bl-sw-fr	3B : 1B : 4L
	wa-ra-fm	3L+2B : 1B : 1L+1B
	wa-ra-vi	2L+2B : 1B : 2L+1B

application performance requirements into account, and chooses the points that minimize total energy consumption from the sorted profiled data, without violating the resource constraints. As discussed earlier, the selected thread-to-core mapping is not altered during the application execution.

In a single-application scenario, there is only one active application. Fig. 4.6 shows a comparison of the adopted approach with existing techniques in terms of energy consumption. First, an energy efficient thread-to-core mapping is determined to satisfy the given performance requirement and resource availability. The experimental observation shows that, for most applications, adopted thread-to-core mapping approach tends to choose all available cores, except for *fr* and *ra* (see single-application scenario in Table 4.4), as it is the energy efficient point. Afterwards, the ITMD's online DVFS approach takes control of the frequency scaling to minimize the wasted cycles in case of memory-intensive workloads. It periodically samples the PMC data and uses a proactive *V-f* setting strategy using the workload prediction. From Fig 4.6, it can be observed that the ITMD outperforms all existing approaches which used HMP scheduler for thread-to-core mapping with various Linux governors for DVFS and *WMI*. Except for *fr* and *ra*, energy savings are mostly due to DVFS as both HMP and ITMD approach have the similar thread-to-core mapping. The higher energy savings in case of *fr* are because of mapping threads to power efficient A7 (L) cores, which is the same as that of WMI. It also has long execution that benefits from periodic DVFS. On average, ITMD approach

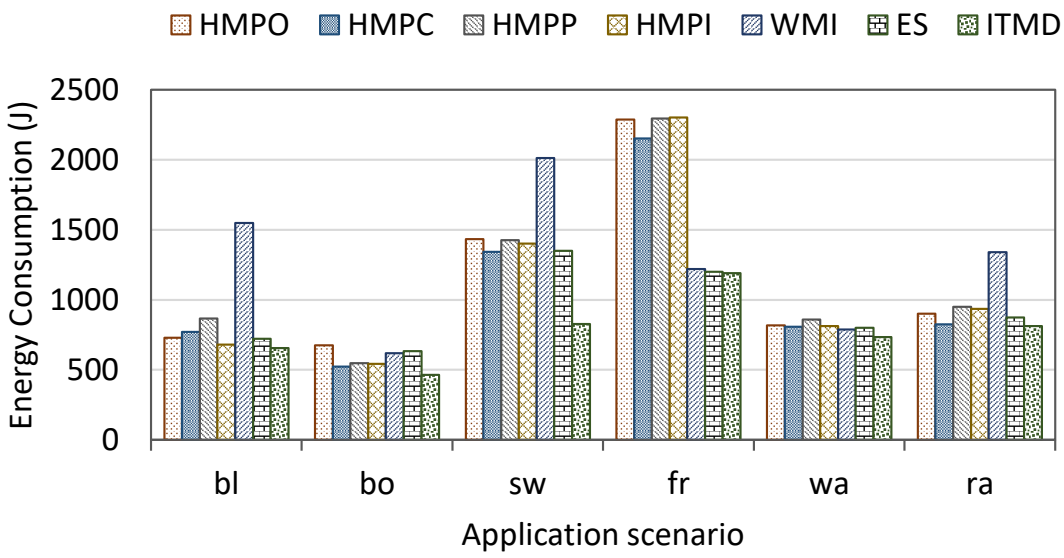

FIGURE 4.6: Comparison of ITMD approach with reported approaches for single active application.

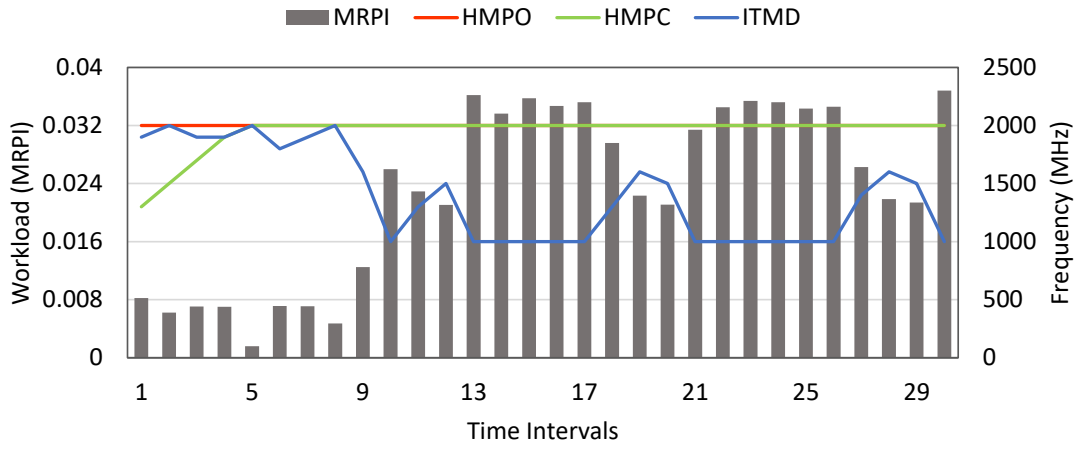

FIGURE 4.7: MRPI and frequency at different time intervals of the application *fr* execution for various approaches.

achieves, 25%, 20%, 27%, 22%, and 33% energy savings while meeting performance requirements compared to *HMPO, HMPC, HMPP, HMPI* and *WMI*, respectively. Furthermore, as ITMD applies online DVFS at regular intervals, it provides better energy savings (17%) than the exhaustive search-based approach (ES).

Moreover, Fig. 4.7 shows the adaptiveness of the proposed online DVFS technique to workload variations for the application *fr*. A high MRPI leads to scaling down the frequency, thereby *ITMD* approach minimizes the power consumption, whereas *HMPO* and *HMPC* runs at max frequency. This is due to the fact that whilst the application is memory-intensive, it places a high load on the processor cores as far as the load measured by the kernel is concerned. Therefore, these select the highest frequency even if it does not offer improvement in performance.

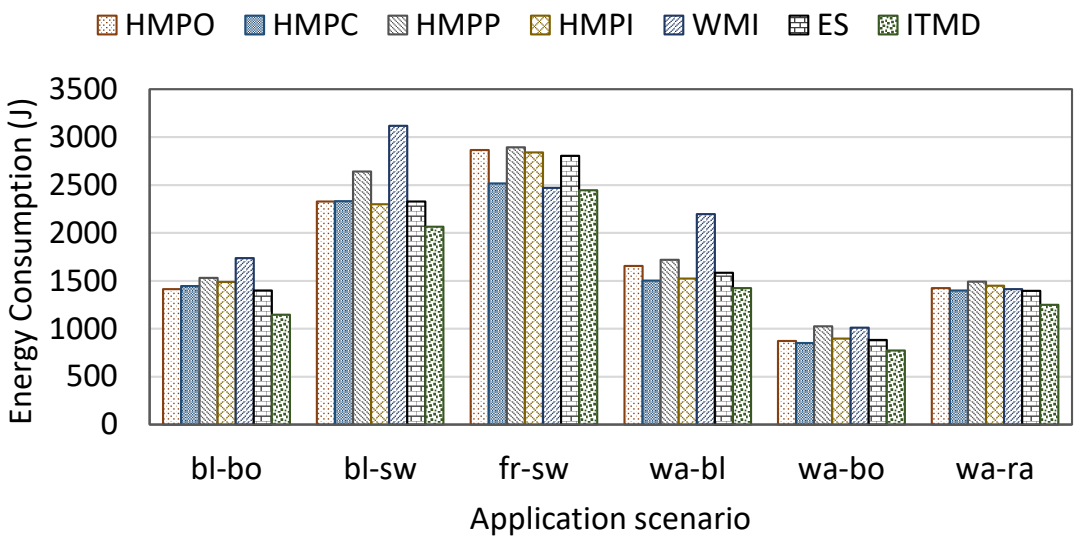

FIGURE 4.8: Comparison of ITMD approach with reported approaches for two active applications.

In case of a multiple-application scenario, at a given moment two or more active applications will be contending for resources to meet their requirements. Such scenarios can be observed in a mobile phone where user tries to run more applications at the same time, e.g., mp3-decoder to listen to music and downloading a file. A set of two applications from Table 4.2 are considered to stress on effectiveness of the adopted approach in choosing resources and *V-f* pair for minimizing the energy consumption while meeting each application performance requirement. Due to limited resource availability and contention, the energy savings are comparatively less than the single-application scenario. Fig. 4.8 gives the energy consumption for various approaches. On average, proposed approach achieves 13%, 14%, 10%, 20%, 15%, and 23% energy savings while meeting performance requirements compared to *ES, HMPO, HMPC, HMPP, HMPI* and *WMI*, respectively. Moreover, chosen resource combinations are presented in row *two* of Table 4.4.

To further validate the ability of the ITMD to adapt to concurrent execution of multiple applications, three-application scenario is also considered. An increase in number of active applications leads to reduced solution space for choosing an energy efficient thread-to-core mapping. As mentioned before, it is caused by the resource constraints (see Table 4.4 for resource combination) and increased contention due to concurrent workloads and demand for meeting their requirements. Furthermore, the online DVFS will have a little choice to scale down the frequency as it has to satisfy the performance requirement of different dynamic workloads (e.g. compute and memory). This results in decreased energy savings compared to single and two-application scenarios. Fig. 4.9 presents the comparison of adopted methodology with various previous techniques. On average, ITMD achieves 11%, 12%, 9%, 16%, 14%, and 30% energy savings while meeting performance requirements compared to *ES, HMPO, HMPC, HMPP, HMPI* and

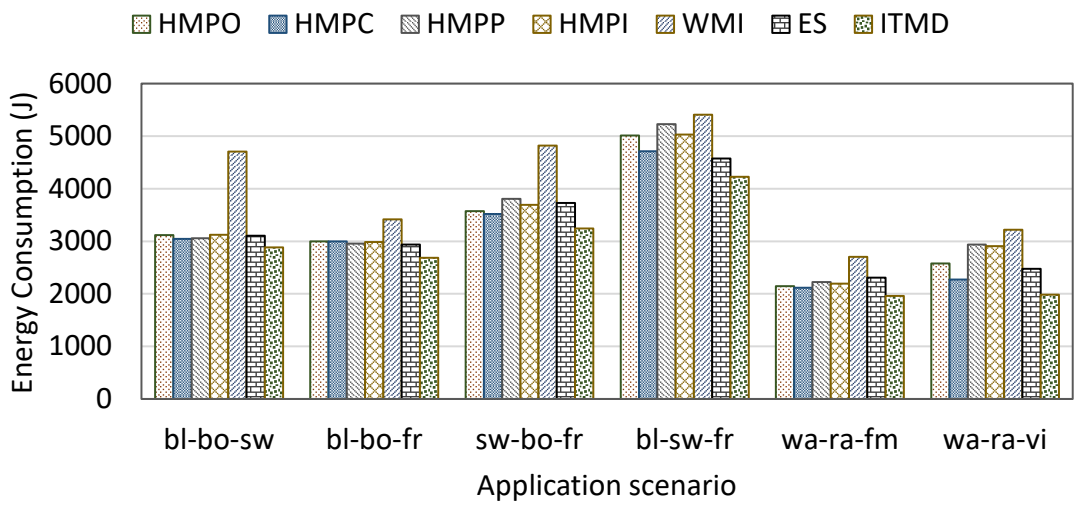

FIGURE 4.9: Comparison of ITMD approach with reported approaches for three active applications.

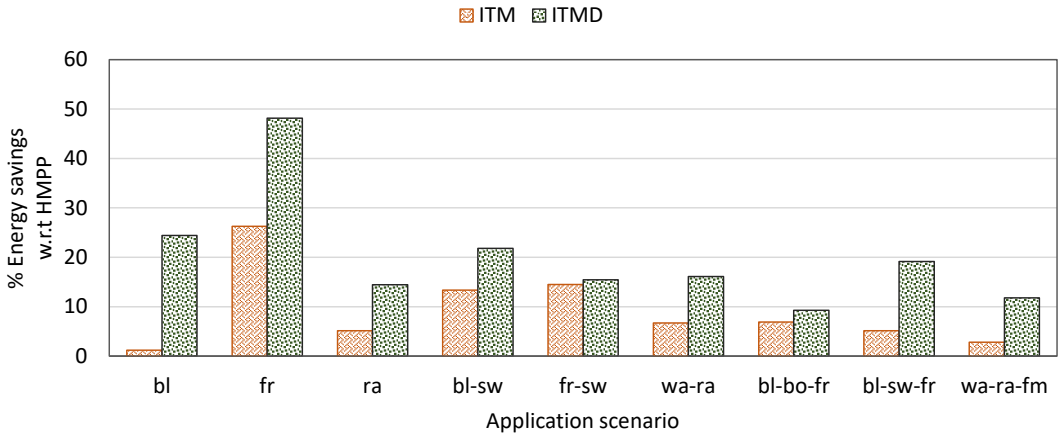

FIGURE 4.10: Percentage of energy savings achieved by proposed *ITM* and *ITMD* respectively.

WMI, respectively.

The four and more applications scenario seems to be not feasible because of high resource contention, leading to not meeting given requirements (some applications were terminated by *out of memory (OOM) killer* daemon when multiple memory-intensive applications are run). It is explained further in the following section. On average, the adopted approach achieves energy savings up to 33% compared to existing techniques.

4.3.1.2 Breakdown of Energy Savings

The individual contribution of the thread-to-core mapping (*ITM*) and online DVFS in energy savings is computed by disabling and enabling DVFS respectively. Further, percentage energy savings are calculated by comparing against the *HMPP*, as shown in Fig. 4.10 for different application scenarios. On average, *ITM* achieves energy savings

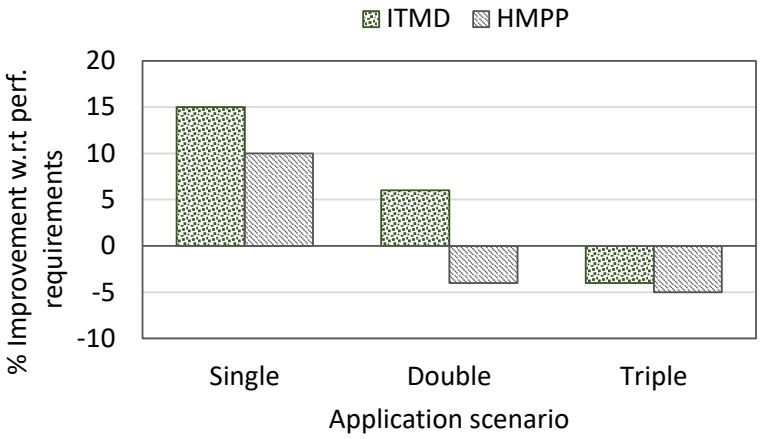

FIGURE 4.11: Performance improvement/degradation of the adopted approach and *HMPP*.

of 9% compared to *HMMP*. Further, when the proposed online DVFS is applied on top of *ITM* (*ITMD*), an extra 11% of energy savings is obtained. It can be observed from Fig. 4.7 that the workload varies over time, for example from low MRPI to high MRPI (at 8^{th} and 10^{th} time intervals). As the online DVFS is applied at regular intervals, the ITMD approach exploits these variations to achieve energy efficiency even for a compute-intensive application.

4.3.1.3 Performance

As discussed earlier, performance requirements are defined for each application. The proposed approach always tries to meet the performance requirement of each application, i.e. finishing the execution within the stipulated time. To validate the adaptability of the ITMD to the performance requirements, the achieved performance is compared against the given performance requirement, computed as percentage improvement, for all the application scenarios. The average percentage improvement in each scenario is presented in Fig. 4.11 in comparison with *HMPP* (performance requirements-unaware), as it maximizes the performance. The figure shows that *ITMD* always outperforms *HMPP* even when there is a high contention due to more active applications (e.g. three-application scenario). Moreover, in some cases, the adopted technique achieves up to 15% improvement over given performance requirements, whereas *HMPP* achieves 10% improvement. Additionally, the following observation can be made from Fig. 4.11. As the number of active applications increases, meeting high performance requirements is not feasible (see three-application scenario in Fig. 4.11) due to resource constraints and interference. Therefore, choosing a low performance requirement or a platform with more resources may guarantee meeting the requirements while running higher number of active applications.

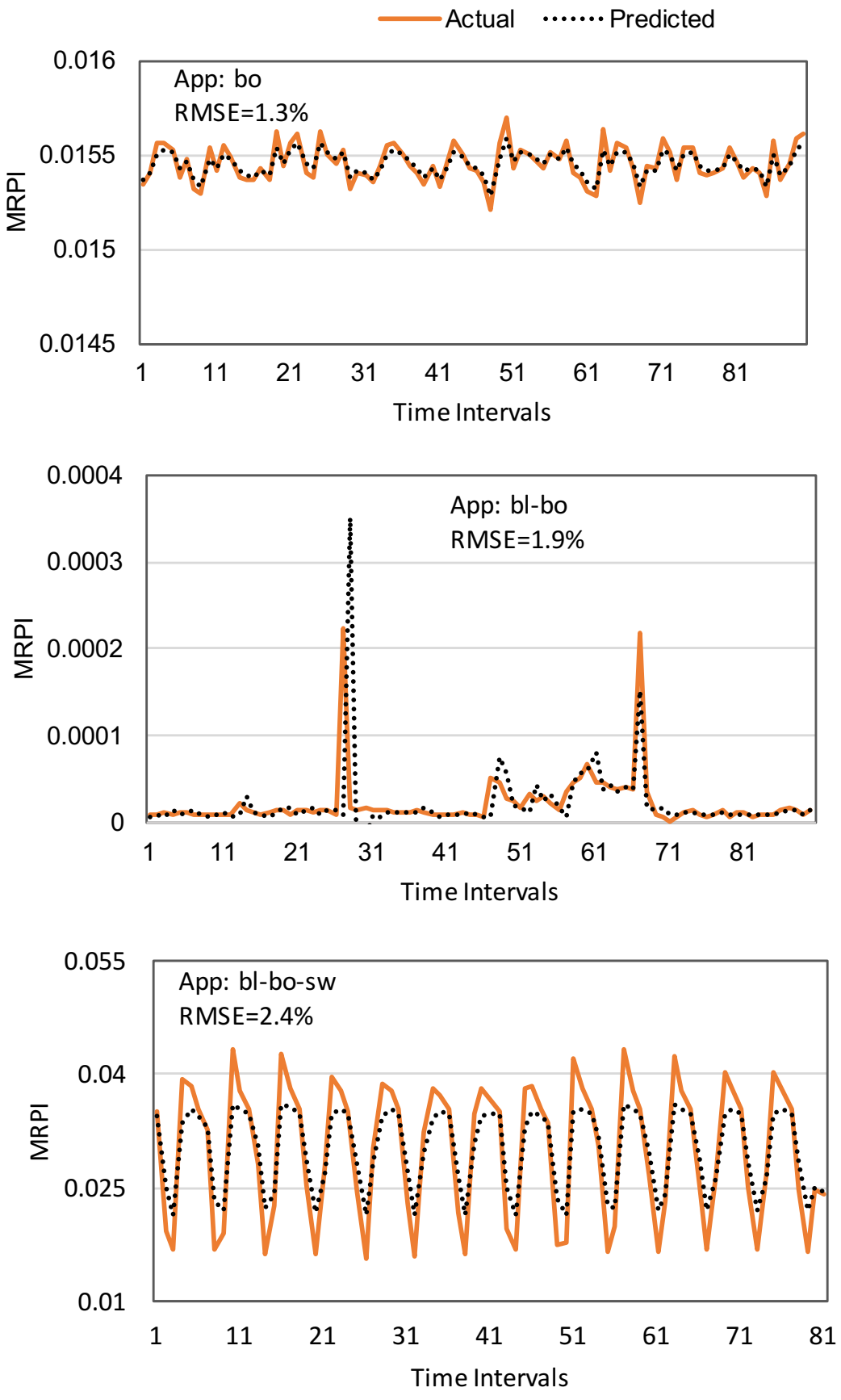

FIGURE 4.12: Workload prediction using EWMA for three different application scenarios - one, two and three active applications (top to bottom).

To further substantiate the need for performance requirements-aware approaches, the number of violations are recorded by disabling the performance requirements-aware property of the proposed approach, resembling the technique presented in [50]. As a result, the mapping algorithm produces thread-to-core mappings that minimize the total energy consumption, which may not satisfy the performance constraints. For single application scenario, the average percentage of performance requirement-violating mappings are nearly zero. This is because, using all the cores (4L and 4B) leads to minimum energy and better performance for all the applications (except for *fr* (4L)). In case of multiple applications executing concurrently, performance violations are significantly high. The average percentage of performance requirement-violating mappings are 98.2% and 99.6% for two- and three-application scenarios, respectively.

4.3.2 Workload Prediction

The accuracy of the predicted workload as compared to the actual workload of the prior time intervals depends on the smoothing factor γ (Eq. 3.4 in Section 3.2.1.3). The optimal value of γ was experimentally obtained by sweeping it between 0.1 and 1, and observing the corresponding workload miss-predictions (under/over) for various application workloads. A value of 0.6 is used for all the experiments as it resulted in relatively accurate workload prediction, similar to [35]. Fig. 4.12 shows the actual and predicted values for three different application scenarios along with the percentage root mean square error (that is up to 2.4%). The figure shows that the prediction error slightly goes up with the number of active applications, which is due to increased dynamic workload variations.

4.3.3 Overheads of the Proposed Approach

4.3.3.1 Runtime Overhead

The runtime overhead of the adopted approach includes time for finding thread-core-mapping (T_{map}) and *V-f* pair ($T_{V\text{-}f}$), which can be represented as,

$$T_o = T_{map} + T_{V\text{-}f} \tag{4.7}$$

$$T_{V\text{-}f} = \frac{T_{ex} - r * len * T_s}{T_s} * [VfS_o + PMC_o + Proc_o] \tag{4.8}$$

where T_o, T_{ex}, r, VfS_o, PMC_o, and $Proc_o$, represent total overhead, execution time, number of times the adaptation is paused, overheads associated with *V-f* switching,

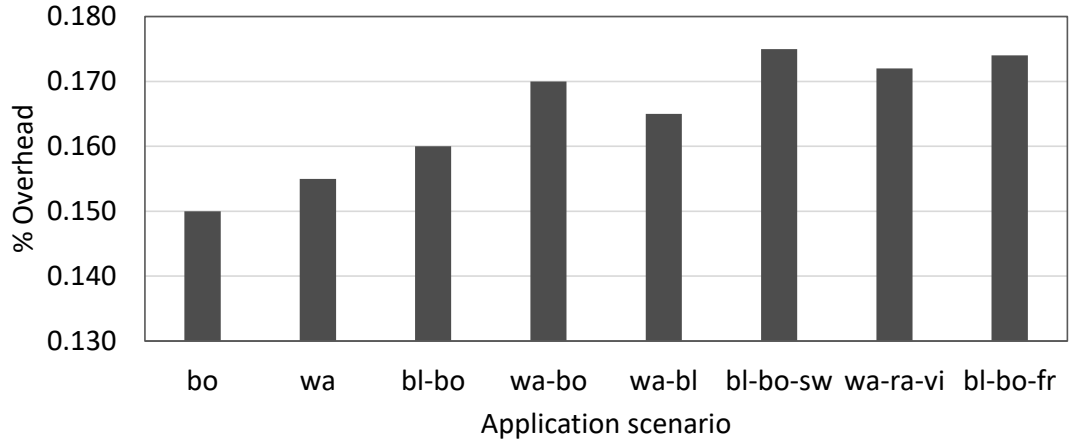

FIGURE 4.13: Runtime overhead of the proposed approach

PMC collection and remaining processing steps (involving *len* and T_s, shown in Algorithm 3), respectively. T_{map} depends on the implementation of the Algorithm 2, in our case it is up to 1.6 ms (averaged over various application scenarios). Moreover, $T_{V\text{-}f}$ is about 300 μs, which is 0.15% of T_s (200 ms). Fig. 4.13 illustrates the total runtime overhead, computed as percentage of total execution time, for eight application scenarios. The runtime overhead for application scenario *bl-bo-fr*, having a long execution time of 1053 sec is ∼0.17%, which is very minimal. Whereas, commonly used pure online learning-based approaches have significant overheads (up to 216 sec for learning and 1 sec for subsequent stages) for a single-application scenario [35]), which gets further aggravated by dynamic workload variations causing frequent re-learning. Therefore, the scalability of such approaches in comparison to the proposed technique is limited for multi-core platforms executing multiple multi-threaded applications concurrently.

4.3.3.2 Offline Analysis Overhead

As discussed earlier, the profiled data of each application contains performance (1/execution time), energy consumption, number of big, and number of LITTLE cores for each design point. The total number of design points for each application is 24, which results in a small storage overhead of 770 bytes. The energy overhead due to storing of profiled data is already included in the energy consumption values reported in the previous sections.

4.4 Discussion

This chapter presented a runtime management system for concurrent execution of multiple multi-threaded applications on heterogeneous multi-core platforms that contain

different types of cores. It first selects thread-to-core mapping based on the performance requirements and resource availability. Then, it applies online adaptation by adjusting the *V-f* levels to achieve energy optimization, without trading-off application performance. For thread-to-core mapping, offline profiled results are used, which contain performance and energy characteristics of applications when executed on the heterogeneous platform by using different types of cores in various possible combinations. For an application, thread-to-core mapping process defines the number of used cores and their type, which are situated in different clusters. The online adaptation process classifies the inherent workload characteristics of concurrently executing applications, incurring a lower overhead than existing learning-based approaches as demonstrated in this chapter. The classification of workload is performed using the metric Memory Reads Per Instruction (MRPI). The adaptation process pro-actively selects an appropriate *V-f* pair for a predicted workload. Subsequently, it monitors the workload prediction error and performance loss, quantified by Instructions Per Second (IPS), and adjusts the chosen *V-f* to compensate. Proposed runtime management approach is validated on a hardware platform, the Odroid-XU3, with various combinations of multi-threaded applications from PARSEC and SPLASH benchmarks. Results show an average improvement in energy efficiency up to 33% compared to existing approaches while meeting the performance requirements.

The next chapter extends this work by incorporating adaptive thread-to-core mapping to improve the adaptability and eliminate the dependency of runtime management system on application offline analysis results.

Chapter 5

Adaptive Mapping and DVFS for Heterogeneous Multi-cores

The previous chapter presented an energy-efficient static thread-to-core mapping and DVFS approach, targeting heterogeneous multi-cores. However, this approach relies on offline application profiling and do not adapt to dynamic application arrival and completion. As detailed in the following sections, such approaches are not efficient in handling unknown applications and exploiting available resources efficiently under varying runtime execution scenarios. Therefore, in this chapter, an approach to runtime management to address the aforementioned problems is considered. As detailed in the Section 2.2, modern mobile platforms based on Multiprocessor System-on-Chip (MPSoC) are containing greater number of heterogeneous cores to support highly diverse and varying workloads (e.g., the Odroid-XU3 [28] and Mediatek X20 [30]). Such platforms often execute applications concurrently, which simultaneously contend for system resources and typically exhibit varying resource demands over time [122]. Each application may have different performance requirements and exhibit various workload phases during its execution [20]. To adapt to such dynamic scenarios, MPSoCs offer an increasing number of resource configurations, such as enabling and disabling cores of different types, defining the thread-to-core mapping for a multi-threaded application, and setting dynamic voltage and frequency (DVFS) operating points.

As discussed in Chapter 4, the process of thread-to-core mapping and setting DVFS levels play a crucial role in exploiting the system properties such that applications can meet their, often diverse, demands on performance and energy efficiency [122]. In general, for each application, the management process first finds a thread-to-core mapping, and then core DVFS level by inspecting the workload profile while satisfying the performance requirement. This problem becomes much more complex when dynamically mapping concurrently executing applications due to contention for resources, and when

the mapping is coupled with DVFS, i.e., energy-efficient allocation of processing cores and selection of DVFS settings [57, 124].

Based on the literature review presented in Section 2.3, the reported approaches for solving above problem fall into three categories: 1) offline, 2) online, and 3) hybrid approaches. Several offline approaches have been proposed targeting different application domains and hardware architectures [43, 108]. These typically use computationally intensive search methods to find the optimal or near-optimal mapping for the applications that may run on the system. Conversely, online approaches [20, 172–174] must not be computationally intensive, as they are required to make efficient application mapping/DVFS decisions at runtime. Therefore, these techniques generally use heuristics to find a suitable platform configuration. Design time approaches usually find solutions of higher quality compared to online techniques, due to extensive design space exploration of the underlying hardware and applications. To address the drawbacks of pure offline and online approaches, various hybrid approaches [5, 10, 19, 40, 50, 52, 108] using offline analysis to make runtime decisions based on the current state of the system are proposed.

However, a review of the prior arts given in Section 2.3, shows that the existing approaches, targeting heterogeneous MPSoCs, have the following shortcomings. They use heavy application-dependent profile data and thus are not efficient in managing dynamic workloads when unknown applications with different performance constraints are executing concurrently. For example, the number of different frequency and core configurations for the Odroid-XU3 platform [28] (four big and four LITTLE cores that can operate at 13 and 19 different frequencies, respectively) is 4080 ((4×13×4×19) + (4×13) + (4×19)). Most importantly, all these approaches do not perform adaptations (changing the mappings and/or DVFS settings) at an application arrival/completion, and performance variations.

To this end, this chapter presents an energy-efficient adaptive mapping approach coupled with DVFS for performance-constrained multi-threaded applications, executing on heterogeneous multi-cores. Our methodology selects an energy-efficient resource combination (number of cores and their type) that meets the application's performance requirement. This is achieved by employing performance prediction models for resource combination enumeration and selection. Furthermore, the application workload, performance and its status (finished or newly arrived) are monitored for adaptive resource allocation and DVFS. The key contributions of this chapter are:

1. A performance prediction model that has a maximum percentage error of 8.1%, which is 7.9% lower than the previously reported model [19].

2. An online energy-efficient mapping approach that allocates processing cores to application(s) based on performance constraints without using any application-dependent offline results.

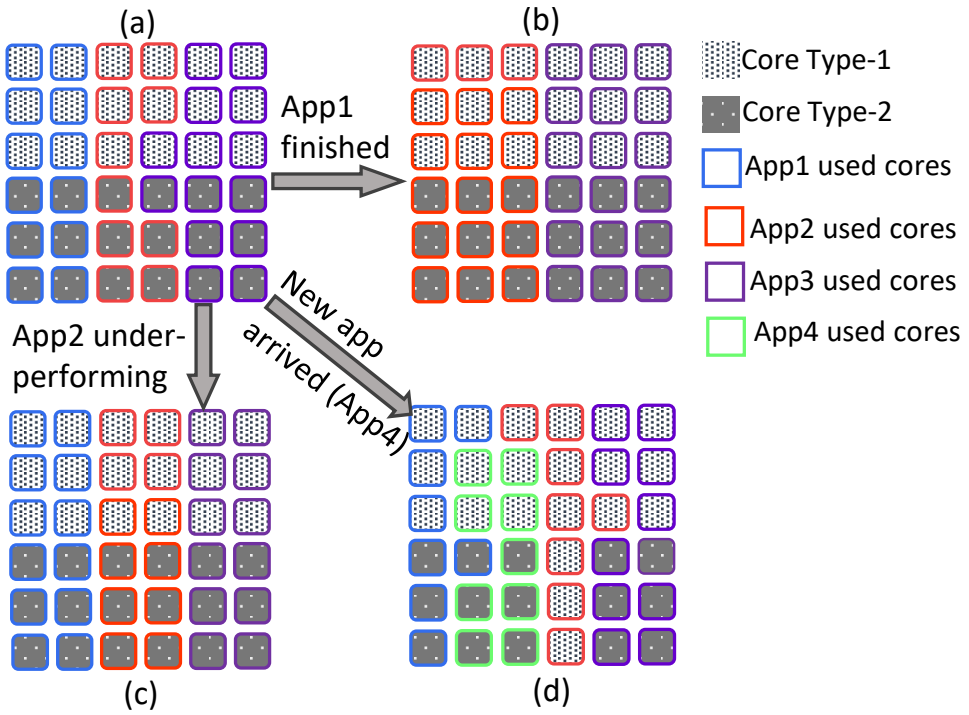

FIGURE 5.1: A motivational example showing three possible runtime execution scenarios (b, c & d) when a system, having two types of cores - Type-1 and Type-2, starts with executing three performance-constrained applications (a). Cores running the same application are encircled with a line of the same color. App1, App2, App3, and App4 represent user applications.

3. To adapt to application arrival or completion times, and workload/performance variations, an adaptive approach that adjusts the existing thread-to-core mappings and DVFS settings during application execution is presented.

4. Experimental validation of the proposed approach on the Odroid-XU3 [28], using several multi-threaded applications from PARSEC [15] and SPLASH [18] benchmarks.

The remainder of this Chapter is organised as follows. Section 5 presents a motivational example for this work, while section 5.1 presents the problem definition for this work. A detailed description of the proposed adaptive mapping and DVFS approach is given in Section 5.2. The experimental setup and validation of the approach are explained in Section 5.3. Finally, Section 5.4 summarises the contributions of this chapter.

Motivation

A heterogeneous computing system with two types of cores, concurrently executing multiple performance-constrained applications concurrently, is illustrated in Fig. 5.1. Dotted squares colored in white/black represent processing cores. For example, such

scenarios could be observed when a smartphone user simultaneously runs a music player, Facebook, background email service, downloading a file, etc. As shown in Fig. 5.1(a), the initial mapping for each application (App1, App2, and App3) is decided based on its performance constraints while considering the energy as an optimization goal. This requires finding an energy-efficient resource combination (number of cores and their type). While these applications are executing, there are primarily three runtime execution scenarios possible: i) any application(s) may finish executing, ii) an application(s) may experience performance degradation due to contention for shared resources, and iii) a new application(s) may arrive into the system. In the first case, if application App1 finishes execution, its resources can be allocated to App2 and App3, which may help them execute faster (and hence put them into a low-power mode sooner), as shown in Fig. 5.1 (b). This may result in increased performance and lower energy consumption, because power is dissipated for a shorter duration.

For case ii), as reported by previous work [53, 124], applications go through different workload phases during their execution. For example, some workload phases could be more compute-intensive than others or vice versa. Furthermore, in case of concurrent execution, an application may experience interference from other applications due to shared resources such as Last Level Cache, Memory, etc. All the factors above culminate into variation in an application's workload, subsequently leading to variation in application performance. Therefore, the application's performance has to be monitored periodically, and appropriate action (changing the DVFS setting or remapping) taken to avoid/minimize performance violations. Fig. 5.1 (c) demonstrates such a case, where more resources are allocated to App2 to mitigate the performance degradation experienced during runtime. If there are no free cores available, as in our case, the cores are taken from the over-performing App3.

For case iii), considering the processing capabilities of the underlying hardware, the user may launch a new application while other applications are running. If all the processing cores have been allocated to the already running applications, the runtime management software should check if there are possibilities to re-adjust the current mapping and allocate resources to the newly arrived application without violating performance constraints. This is shown in Fig. 5.1 (d), where App4 is added to the system while App1, App2, and App3 are executing. The resources of over-performing applications App1 and App3 are allocated to App4 while keeping the same number of cores for App2.

As discussed before, existing approaches do not consider the above execution scenarios (case i, ii and iii) for adaptation and moreover, they also depend on extensive offline characterisation and/or instrumentation of the chosen applications. As experimentally demonstrated in Section 5.3, adaptation at application arrival and completion, and workload/performance variations would lead to better utilisation of the system resources, and higher energy efficiency and performance.

5.1 Problem Formulation

Earlier studies have shown that the thread-to-core mapping problem alone is NP-complete [122]. Therefore, combining it with DVFS would increase the complexity of mapping problem due to the huge design space, thereby making the runtime management significantly inefficient. Similarly, if the number of cores or heterogeneity or frequency levels increases, the design space becomes too large for solving at runtime and even for offline analysis [124]. To address this, as per literature, thread-to-core mapping and DVFS are considered separately to minimize the runtime overheads. The following forms the problem definition.

Given a set of performance constrained applications to be executed concurrently or at different moments of time on a heterogeneous MPSoC supporting DVFS.
Find an initial energy-efficient thread-to-core mapping for each application and then apply DVFS and/or adaptive remapping at runtime to minimize the energy consumption, if any of the following occur:

- An existing application finishes or a new application arrives into the system
- The performance constraints of any running applications are violated
- The workload of an application varies during execution (e.g., from compute-intensive to memory-intensive)

Subject to meeting the performance requirement of each application without violating the resource constraints (number of available cores in the platform)

5.2 Adaptive Thread-to-Core Mapping and DVFS

This section presents a detailed discussion of the proposed dynamic thread-to-core mapping and DVFS methodology. An outline of the proposed approach is presented in Fig. 5.2 and corresponding pseudocode in Algorithm 4 and 5. The figure also shows the proposed approach as a runtime manager, sitting along side of the OS in a three-layer representation of a multi-core system. The proposed approach mainly contains the following steps:

1. Runtime Data Collector (Section 5.2.1.1)
2. Performance Predictor (Section 5.2.2)
3. Resource Combination Enumerator (Section 5.2.3)
4. Resource Selector (Section 5.2.4)
5. Resource Allocator/Reallocator (Section 5.2.5.1)

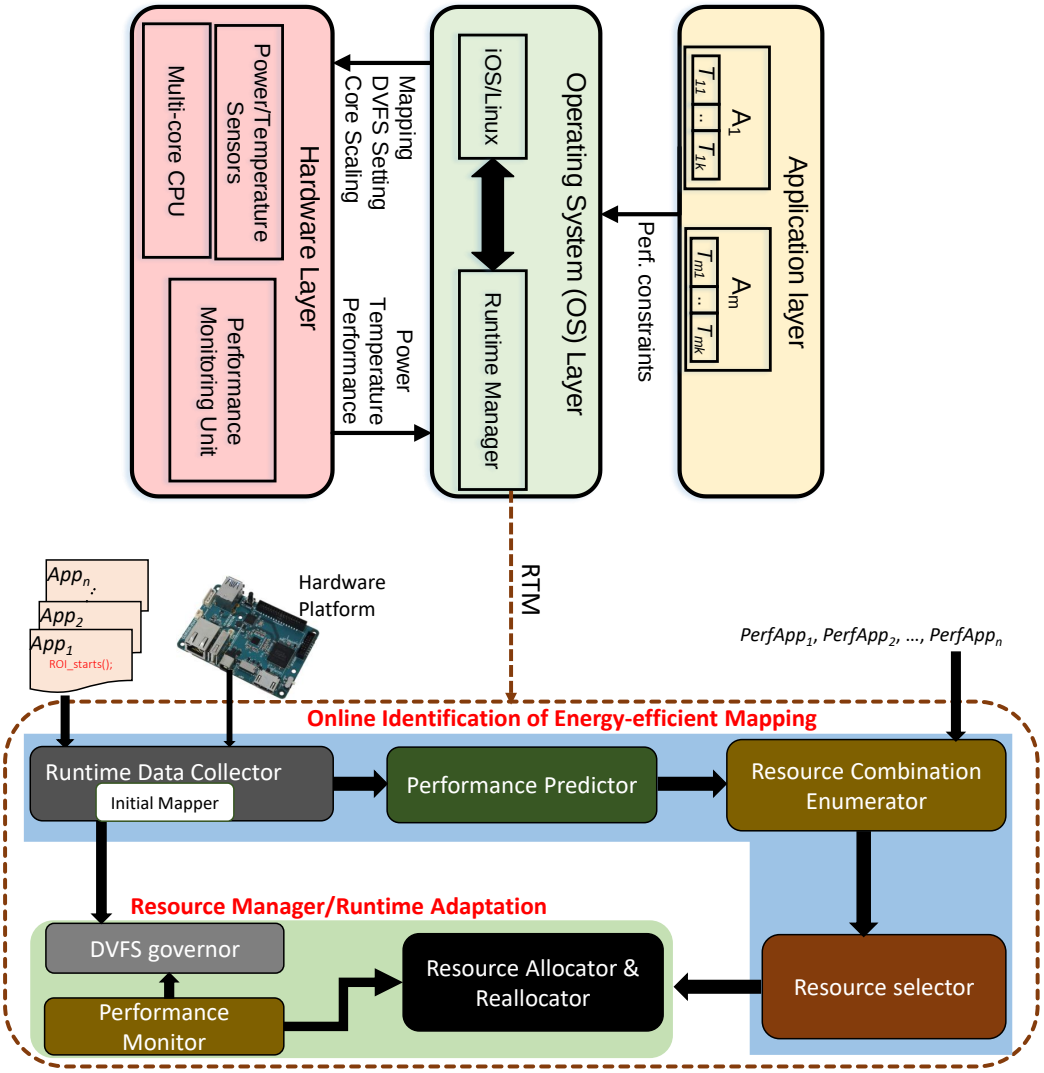

FIGURE 5.2: Illustration of various steps in the proposed adaptive thread-to-core mapping and DVFS approach (bottom) and showing its placement in the three-layer representation of a multi-core platform (top).

6. Performance Monitor (Section 5.2.5.2)

7. DVFS Governor (Section 5.2.5.3)

The arriving performance-constrained applications are added to the queue, called `Apps`, and the *initial mapper* allocates a processing core to each application in the queue. Meanwhile, the *Runtime Data Collector* periodically gathers necessary runtime information through performance monitoring counters (PMCs) for the performance predictor, DVFS governor and performance monitor. The *Performance Predictor* estimates the application performance, using instructions per cycle (IPC) or instructions per second (IPS), on various types of cores by using the runtime information collected on a single type of core. The estimated performance of an application on various types of cores is then utilised for enumerating the list of resource combinations (the number of cores and

TABLE 5.1: Parameters used in the proposed approach

Number of Active Cores
Frequency of the Cores
L1 I-Cache Misses
L1 D-Cache Misses
L2 Cache Misses
Instructions Retired
Branch Misses
CPU Cycles
Per Core CPU Utilisation
Memory Reads Per Instruction

their type) that meet the performance constraints of the application (*Resource Combination Enumerator*). Next, the *Resource Selector* picks the resource combination that would lead to lower energy consumption. Finally, the *Resource Manager* keeps track of the variation in application performance, workload and completion/arrival time to decide on allocating cores and voltage-frequency settings. The following discusses each step of the proposed methodology in detail.

5.2.1 Online Identification of Energy-efficient Mapping

The proposed approach first identifies an energy-efficient thread-to-core mapping for each performance-constrained application in a concurrent execution scenario without using offline profiled results. This process involves the following steps.

5.2.1.1 Runtime Data Collector

The proposed approach requires various parameters for making runtime decisions while concurrent applications are executing on the platform. These parameters are collected by the Runtime Data Collector. The list of parameters used in this work is given in Table 5.1. Of these parameters, `CPU Cycles`, `Instructions Retired`, and `L2 Cache Misses` are periodically collected to measure Memory Reads Per Instruction (MRPI), per core CPU Utilisation, and IPC or IPS for detecting the workload and/or performance variations by the DVFS governor and Performance Monitor (details are given in Section 5.2.5). The performance monitoring unit (PMU) of the processor is initialized to monitor the above parameters through the routine `PMU_initialize()` (line 1, Algorithm 4). Note that all the parameters are collected only when an application(s) arrives into the system, which are used by the Performance Predictor. When an application arrives, the *Initial Mapper* adds it to the application queue and allocates a free core to the application to start application execution (lines 3-9, Algorithm 4). As application execution begins with the serial section, the initial mapper tends to allocate a big core to the application. However, if an application's serial section is memory-intensive, measured by MRPI, the

application is migrated to a LITTLE core as it results in a greater power efficiency [175] (line 10, Algorithm 4). Data collection starts in the region of interest (ROI) (indicating the parallel code in the application) as that is when actual computation starts and the benefit of allocating more than one processing core can be seen [15]. This is accomplished by notifying the Runtime Data Collector through the `ROI_starts()` routine when the ROI of an application starts (lines 12-15, Algorithm 4).

5.2.2 Performance Predictor

To allocate resources in a heterogeneous multi-core system to meet the performance requirements of an application, it is essential to know how the application performs on various types of cores [175]. This can be achieved either by executing the application on all types of cores in a platform or by estimating the performance of application for different types of cores by running only on one core type. The former approach requires the migration of the application across various core types. As observed experimentally in [175], migration cost across clusters on a big.LITTLE architecture is relatively high: 2.10 ms to move a thread from a LITTLE cluster to a big cluster, and 3.75 ms to move from a big cluster to a LITTLE cluster. This overhead grows with the number of cores and types. Considering the runtime overheads and scalability, this is not an efficient approach. However, this approach would not need offline analysis as everything is measured at runtime. On the other hand, a performance prediction-based approach avoids thread migration by using the performance models built offline or online. Previous approaches have shown that learning performance models at runtime would make the approach non-scalable and has its overheads in terms of execution time and power [50, 124]. Therefore, the performance models are built at design time through a generalized methodology, which can easily be adopted to a new platform/architecture.

Performance models

Application performance is usually measured in terms of IPS or IPC, and the relative improvement in the performance is referred to as *speedup*. The speedup η is defined as follows,

$$\eta = \frac{IPC_{CoreType1}}{IPC_{CoreType2}} \tag{5.1}$$

where, $IPC_{CoreType1}$, $IPC_{CoreType2}$ are the IPC of the application achieved on core type-1 and core type-2, respectively. The performance model estimates the speedup, which is used for computing the application performance on a second core type ($IPC_{CoreType2}$),

Algorithm 4 Proposed Dynamic Thread-to-core Mapping

Input: Applications and performance constraints (`Apps`)
Output: ∀`Apps`, mappings and DVFS settings

```
1: PMU_initialize()  // initialises PMCs
2: while (1) do
3:     if (NewApp) then
4:         Update the Application Queue 'Apps';
5:         NewApp = 0;
6:     end if
7:     for ∀i ∈ Apps do
8:         if (unmapped) then
9:             Allocate a free core 'l' to 'i' and execute;
10:            Measure MRPI and move onto an appropriate core (j);
11:            /*Data collection for performance model*/;
12:            Wait until ROI begins;
13:            pmcs = pmcs_data_collect(j);
14:            f = cpufreq_get_freq_hardware(j);
15:            pmcs.push_back(f);
16:            η = speedup_estimate(pmcs,j);
17:            Compute possible resource combinations and energy-efficient resource
    combination t_h (Eq. (5.4), (5.5) & (5.6));
18:            Allocate resources as per t_h;
19:        end if
20:    end for
21:    /*Distribute the free cores to active applications*/
22:    Sort the applications by η (list);
23:    while (freecores>0) do
24:        Increase the resources of app i ∈ list by y;
25:        freecores = freecores - y;
26:        i++;
27:    end while
28:    /*Application performance and workload adaptation*/
29:    If application workload changes call DVFS(); //Algorithm 2
30:    for i ∈ Apps do
31:        if App 'i' under-performs then
32:            Increase frequency or allocate more cores;
33:        end if
34:    end for
35:     /*Application completion detection and adaptation*/
36:    if p ∈ Apps finishes then
37:        Distribute freed resources of 'p' to under-performing apps;
38:        Allocate remaining resources to apps equally by sorting them based on η;
39:    end if
40:    /*if stop_governor is set, process exits*/
41:    if (stop_governor) then
42:        PMU_terminate(); //Terminates PMC collection
43:        exit(0);
44:    end if
45: end while
```

by running the application on one core type and collecting the runtime parameters, and measuring its performance ($IPC_{CoreType1}$) (line 16, Algorithm 4).

To build the performance models, three steps are followed. The first step is identifying the parameters/metrics that capture the most performance-limiting factors by analysing the correlation between various metrics and speedup. Modern processors support monitoring of various architectural events which can be used for analysing the performance, power, etc. However, not all metrics that contribute to performance can be monitored simultaneously due to the limited number of hardware PMCs provided by the platform. For example, on an Odroid-XU3/XU4, the Cortex-A15 processor allows monitoring of seven events, including the cycle counter, at a time. Therefore, metrics that contribute more to the speedup have to be identified. Based on analysis and the information given in [175, 176], we have identified that cache misses (L1 I/D-Cache & L2 Cache), branch misses, CPU cycles and instructions retired are the appropriate PMCs for estimating the speedup on the chosen platform (listed in Table 5.1). The second step is the collection of characterisation data for a diverse set of applications. As part of this, a diverse set of workloads is created, containing single and multi-threaded applications from SPEC CPU2006 [177], LMBench [136], RoyLongbottom [135], PARSEC 3.0 [15], SPLASH [18], and MiBench [132]. The Odroid-XU3 platform has four Cortex-A7 and four Cortex-A15 cores that can operate at 19 and 13 different DVFS levels, respectively.

For each application, data has been collected for all available frequencies on the platform. Furthermore, in the case of multi-threaded applications, the number of threads/cores are varied from one to four (number of available cores for each type). In each case, six PMCs, frequency of the big and LITTLE CPUs, execution time of the application on the big cluster and LITTLE cluster, and the number of active cores, are all used in the modelling. For consistent results, each experiment is repeated ten times, and corresponding average values are considered while create the model. To create a more general approach for deriving performance models, several statistical and machine learning techniques are explored. Using the open source WEKA workbench [137] to verify the relationship between input features/attributes and output/target variables. As part of the exploration for building performance models, various machine learning methods have been considered. This includes simple linear regression, linear regression, multilayer perception, SMOreg, additive regression, stacking, random forest, and random tree. Of all the explored methods, ensemble approach – additive regression of decision stumps, using boosting for a regression problem, resulted in good accuracy as shown in Section 5.3.2. The runtime data is heterogeneous as the selected parameters in Table 5.1 represent the different activity of the processor. Therefore, additive regression helps in boosting the accuracy of the prediction models by employing multiple models targeting different regions of the data. Moreover, earlies studies also showed that performance of additive regression exceeds or meet that of various other boosting algorithms [178].

The problem of function estimation usually consists of a random *output* variable y and a set of random *input* features $X = \{x_1, x_2, ..., x_n\}$. Given a training sample $\{y_i, X_i\}_1^N$ of known (y, X) values, the objective is to identify a function $\hat{f}(X)$ that relates X to y, such that the expected value ($E_{y,X}$) of some specified error function $\psi(y,\ f(X))$ is minimized.

$$\hat{f}(X) = arg \min_{f(X)} E_{y,X} \psi(y, f(X)) \tag{5.2}$$

In general, boosting approximates $\hat{f}(X)$ by an additive expansion of the form, i.e., adding a set of base learners [179], as shown below:

$$f(X) = \sum_{k=0}^{M} \alpha_k h(X; \beta_k) \tag{5.3}$$

Here, the base learner functions $h(X; \beta)$ are simple functions of X with parameters β = $\{\beta_1,\ \beta_2,\ ...,\ \beta_M\}$ and $\{\alpha_k\}_0^M$ are expansion coefficients. Owing to simplicity, decision stump (one-level decision tree) is used as a base learner in this work. In brief, additive regression takes an initial guess for the speedup (the average speedup observed by all applications in the training set) and estimates the speedup by summing positive and negative additive-regression factors to $f_0(X)$. Each additive-regression factor is associated with an input feature the factor depends on. As the base learner is a decision stump, the input feature is associated with two regression factors, i.e., each of $\{h(X; \beta_k)\}_0^M$ produces one positive/negative additive-regression factor depending on the value of the input feature. Additive-regression factors are computed in a forward stage-wise manner to minimize the squared error of the predictions after M iterations, which decides the number of base learners. Readers can refer to [179] for more details on additive regression.

5.2.3 Resource Combination Enumerator

For each application, a set of all possible resource combinations (number of cores and their type) meeting performance constraints has to be computed to choose the one that minimizes the overall energy consumption (line 17, Algorithm 4). Let R be the set of possible resource combinations on a platform, and $PerfApp_i$ is the performance constraint for an application App_i, then the performance meeting thread-to-core mappings (T_{map_i}) can be defined as follows:

$$T_{map_i} = \{r \in R \mid \texttt{perf}(r) \geq PerfApp_i\} \tag{5.4}$$

Here, $\texttt{perf}(r)$ defines the performance of an application when executed on the resource combination r. For simplicity, let us take the chosen platform, the Odroid-XU3, with two types of cores: big (B) and LITTLE (L); N_b and N_l are set of big and LITTLE cores, respectively. Then, $\texttt{perf}(r)$ is computed as:

$$\texttt{perf}(r) = n_b \times \eta \times IPC_l + n_l \times IPC_l + IPC_o \tag{5.5}$$

where, $\eta = IPC_b/IPC_l$, performance on the big and LITTLE core is denoted by IPC_b and IPC_l, respectively. Furthermore, $n_b \in N_b$, $n_l \in N_l$ and $r = n_l \cup n_b$. IPC_o is the performance overhead incurred when an application is mapped onto cores that do not share a cache. For instance, the big and LITTLE clusters in the Odroid-XU3 do not share caches, which results in an inter-cluster communication overhead when the threads of an application run on both the big and LITTLE clusters.

5.2.4 Resource Selector

The job of *resource selector* is to minimize the energy consumption by selecting an energy-efficient resource combination from the performance meeting thread-to-core mappings $T_{map_i} = \{3L, 4L, 1L + 1B, ...\}$, where L and B refers to big and LITTLE cores, respectively. This can be achieved by selecting a thread-to-core mapping $t_h \in T_{map_i}$ that has the highest performance per watt (PPW) (line 17, Algorithm 4).

$$t_h = arg \max_{t \in T_{map_i}} PPW(t) \tag{5.6}$$

where, $PPW(t)$ is computed as the ratio between IPC achieved for the resource combination '$t \in T_{map_i}$' and its power consumption. This requires measuring the power consumption using on-chip power sensors or employing a power model when a platform does not have power sensors [180]. However, the power model would also require the collection of various PMCs data at regular intervals of time, and its PMCs may be different than the ones used by performance models [175]. This would need multiplexing PMCs, leading to runtime overheads. To address this, the estimated speedup η is used as a proxy for identifying the energy-efficient resource combination when power sensors are not available. This is achieved by choosing a resource combination with the ratio

between the minimum number of big cores to the minimum number of LITTLE cores (C_r) is higher/close to the speedup. This would lead to efficient utilisation of big cores and supports the balanced execution of an application. For example, if the speedup of an application is 2×, then the algorithm initially tends to allocate 2-big cores and 1-LITTLE core. This is also demonstrated in Fig. 5.3, where unbalanced execution of applications `Blackscholes` (top) `Bodytrack` (bottom) resulted in increased execution time and energy consumption. The unbalanced execution is highlighted by arrows. This figure also shows that applications with a speedup greater than one can benefit in terms of energy and performance from allocating more number of cores, as C_r reaches one or higher.

Furthermore, if η is less than 1, all LITTLE cores are allocated as the application does not benefit from executing on big cores in terms of performance/power. This makes the proposed algorithm effective for single-threaded applications as well, where it maps memory-intensive applications ($\eta \leq 1$) onto LITTLE cores, and compute-intensive ($\eta > 1$) onto big cores. Finally, the output of the resource selector is an energy-efficient resource combination and minimum resources that are required for meeting the performance constraints. The information about minimum resources is used by the resource manager.

5.2.5 Resource Manager/Runtime Adaptation

The *Resource Manager*, shown in Fig. 5.2, is responsible for adapting to application arrival/completion, performance/workload variation, and managing resources at runtime. It consists of the Resource Allocator/Reallocator, Performance Monitor and DVFS governor. These are discussed in detail in the following sections.

5.2.5.1 Resource Allocator/Reallocator

The Resource Allocator manages finding free cores and allocating them to the application based on its selected resource combination (line 18, Algorithm 4). This is done by keeping track of allocated cores and free cores available in the platform. The allocated cores are maintained per application, which are used by the performance monitor for measuring application performance and for releasing the resources when the application finishes. While allocating the resources to an application, the resource allocator keeps the knowledge of cores that are leading to over-performance of an application, called extra cores. After finishing the allocation of resources to the applications in application queue (`Apps`), if there are still free resources available, these are allocated to the running applications if the energy consumption can be minimized by reducing the application execution time. The allocation of extra resources is done by first creating a sorted list of active applications in descending order of their speedup. Then, application `i` at the

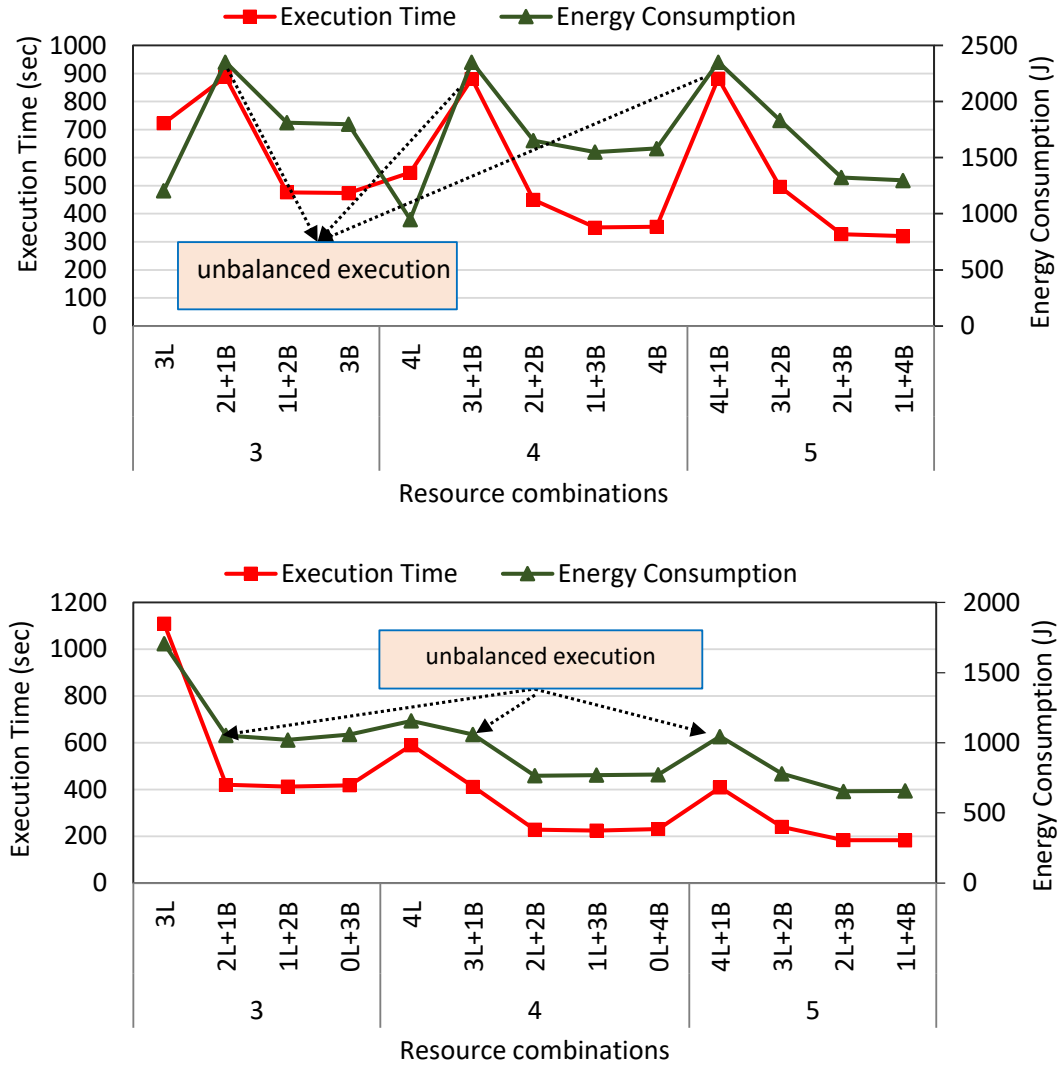

FIGURE 5.3: Energy and execution time at different resource combinations of big (B) and LITTLE (L) for the application `Blackscholes` (top) `Bodytrack` (bottom) and from PARSEC [15], executing on the Odroid-XU3.

top of the list is selected, and its allocated cores are increased by one. This process is repeated for remaining applications in the list until no free cores are left (lines 22-27, Algorithm 4). Note that applications with $\eta < 1$ in the list are given only LITTLE cores as they do not benefit from big cores in terms of energy efficiency.

The Resource Reallocator keeps track of application completion and arrival of new applications into the system. When an application completes execution, it invokes the reallocation routine after releasing the allocated resources (lines 36-39, Algorithm 4). The reallocation routine then distributes the freed resources to the active applications. First, it measures the performance of each application (IPC or IPS) to check if any application is under-performing, i.e., measured performance is lower than the given performance constraint. If an application is under-performing, it then computes the amount of performance loss (the difference between achieved performance and given performance

Algorithm 5 DVFS governor (`DVFS()`)

1: $MRPI_p = 0$, $util_p = 0$, $e_m = 0$, $e_u = 0$;
2: */*Per-core DVFS supporting platforms*
Input: for each core 'i', $MRPI[i]$ and $f_{req}[i]$
Output: voltage-frequency ($V\text{-}f[i]$) for next epoch
3: pmcs = `get_pmc_data`(i);
4: compute actual MRPI ($MRPI_a$) = $\frac{instructions\ retired}{L2\ cache\ misses}$
5: compute actual utilisation ($util_a$) = $\frac{active\ CPU\ cycles}{TotalCPUcycles}$
6: $MRPI_p$ = `predict_mrpi`($mrpi_p$, $mrpi_a$, e_m);
7: MRPI prediction error (e_m) = $mrpi_a - mrpi_p$;
8: $util_p$ = `predict_utilisation`($util_p$, $util_a$, e_u);
9: utilisation prediction error (e_u) = $util_a - util_p$;
10: $V\text{-}f[i]$ = `bin_classify`($util_p$, $mrpi_p$);
11: **if** ($V\text{-}f[i] < f_{req}[i]$) **then**
12: $V\text{-}f[i] = f_{req}[i]$;
13: `cpufreq_set_frequency`(i, $V\text{-}f[i]$);
14: **end if**
15: */*cluster-wide DVFS supporting platforms*/*
16: **for** each cluster 'j' **do**
17: Measure MRPI and utilisation of each core $i \in j$;
18: Compute the minimum MRPI ($mrpi_a$) and utilisation ($util_a$);
19: Repeat steps 6 to 13.
20: **end for**

constraint), and then estimates the required resources using Eq. 5.5 to compensate it. If any resources are remaining after allocating the freed resources to under-performing applications, these resources are distributed among the applications as described in the previous paragraph. Furthermore, when a new application arrives into the system, the resource reallocator tries to identify and allocate the resources as per t_h (Eq. 5.6). This is done by checking if there are enough free resources available in the platform to satisfy the application requirements. In case free resources are not available for meeting performance constraints, the extra cores of over-performing applications are used. After doing this, if the application requirements are still not met, application execution is continued using the available resources until any running application completes and releases allocated resource.

5.2.5.2 Performance Monitor

Applications usually exhibit varying workload profiles (e.g., compute-intensive to memory-intensive and vice versa) during execution. When multiple applications are executing simultaneously, the workload profile of each application gets affected due to contention on shared resources [53]. As a result of this, application performance varies over time, and may lead to the violation of performance constraints. To address this, each application's performance is periodically monitored to detect and compensate when performance constraint is violated (line 30-34, Algorithm 4). An application performance

is measured by collecting PMCs corresponding to instructions retired and CPU cycles on all the cores that the application is currently running on. When an application's performance constraint is violated, either the operating frequency is increased, or more cores are allocated. Raising the operating frequency is given priority over assigning more cores as the latter incurs a migration overhead which is relatively large compared to the DVFS transition latency [175]. The operating frequency is increased in steps of 200 MHz until the performance constraint is satisfied and this frequency (f_{req}) is communicated to DVFS governor (discussed in the next section) to make sure it does not scale down the frequency below this value. After the above step, if any of the applications are still under-performing, as the last solution, more cores are allocated from the available free cores or extra cores of over-performing applications. This allocation is done by computing the performance loss and corresponding required cores using Eq. 5.5. As already explained in Section 5.2.5.1, for applications with $\eta < 1$, LITTLE cores are preferred over big cores.

5.2.5.3 DVFS Governor

Applications go through different workload phases (e.g., compute-intensive, memory-intensive, etc.) and this necessitates choosing a different frequency for each workload phase to reduce the power consumption while maintaining application performance within the bounds. For example, a memory-intensive workload can be executed at a lower frequency than a compute-intensive workload with no/negligible performance loss [53]. To this end, the technique proposed in the Section 3.3 of Chapter 3 is adopted and modified to take f_{req} into account. By doing this, DVFS governor avoids scaling down the frequency below f_{req}, thereby ensuring the applications meeting their performance constraints. Algorithm 5 presents the pseudocode of the DVFS governor. This approach employs a binning-based approach with two classification layers (line 10). The first layer, consisting of utilisation bins, classifies the compute-intensity, and the second layer classifies the memory-intensity using MRPI bins. The classification bins are computed through an offline analysis of 81 diverse workloads, including: 25 from SPEC CPU2006 [177], 20 from LMBench [136], 11 from RoyLongbottom [135], 11 from PARSEC 3.0 [15] and 14 from MiBench [132]. For each application, offline profiling data consisting of MRPI, utilisation and application performance ($\frac{1}{Execution\ time}$) are collected at different DVFS settings available on the chosen platform. The collected utilisation and MRPI for various applications are then grouped into utilisation bins and MRPI bins, and a corresponding voltage-frequency setting is assigned to each bin of the second classification layer. At runtime, the DVFS governor measures the MRPI and utilisation and uses workload prediction to set an appropriate voltage-frequency level (lines 3-9). Furthermore, it can manage both per-core (lines 2-14) and cluster-wide DVFS platforms (lines 15-20). For more details on binning-based DVFS approach, please refer to Section 3.3.

5.3 Experimental Results

This section presents the details of the experimental setup, covering the platform, benchmark applications and reported approaches considered for the comparison. Furthermore, an evaluation of the performance prediction models and benefits of the proposed approach over the previous approaches are discussed, including associated overheads.

5.3.1 Experimental Setup and Implementation

We use the Odroid-XU3 [28], running Ubuntu OS with kernel version 3.10.96 (more details in Section 2.4.1, Chapter 2). Energy consumption is computed as the product of average power consumption (dynamic and static) and application execution time. This includes both the core and memory energy consumption of all the software components, including the proposed approach, OS, applications and other background processes. To evaluate the proposed approach, applications – Blackscholes (bl), Bodytrack (bo), Swaptions (sw), Freqmine (fr), Vips (vi), Water-Spatial (wa), Raytrace (ra), fmm (fm)) – from popular benchmark suites, such as PARSEC 3.0 [15] and SPLASH [18], are taken (more details about these benchmarks are given in Section 2.4.2). These applications exhibit different memory behavior, data partitions, and data sharing patterns. For consistent results, the results are collected by running each scenario ten times and finally, their average values (energy and performance) are computed for the comparison. Different execution scenarios – single application, concurrent execution of multiple applications, dynamic addition of application(s) at runtime – are also considered to mimic the real-world behavior. To ensure the deterministic execution of application and to meet its performance constraint, no two applications share the same cores. However, the threads of the same application share the allocated cores to maximize resource utilisation.

The proposed approach is implemented as a *user space* application by using the `Perfmon2` [167] and `cpufrequtils` framework. `Perfmon2` enables the *user space* access to the performance monitoring unit (PMU), and `cpufrequtils` helps in setting/getting the operating frequencies. Standard Linux API (`sched_setaffinity(2)`) is used to control the CPU affinity of processes, i.e., to bind the applications to specific cores. The thread-to-core mapping algorithm operates at a coarser granularity (500 ms) considering its higher migration overhead. As the workload of application changes randomly, to capitalize on these changes for energy savings, the DVFS governor is operated at a finer granularity of 200 ms as per the observations made in previous chapters.

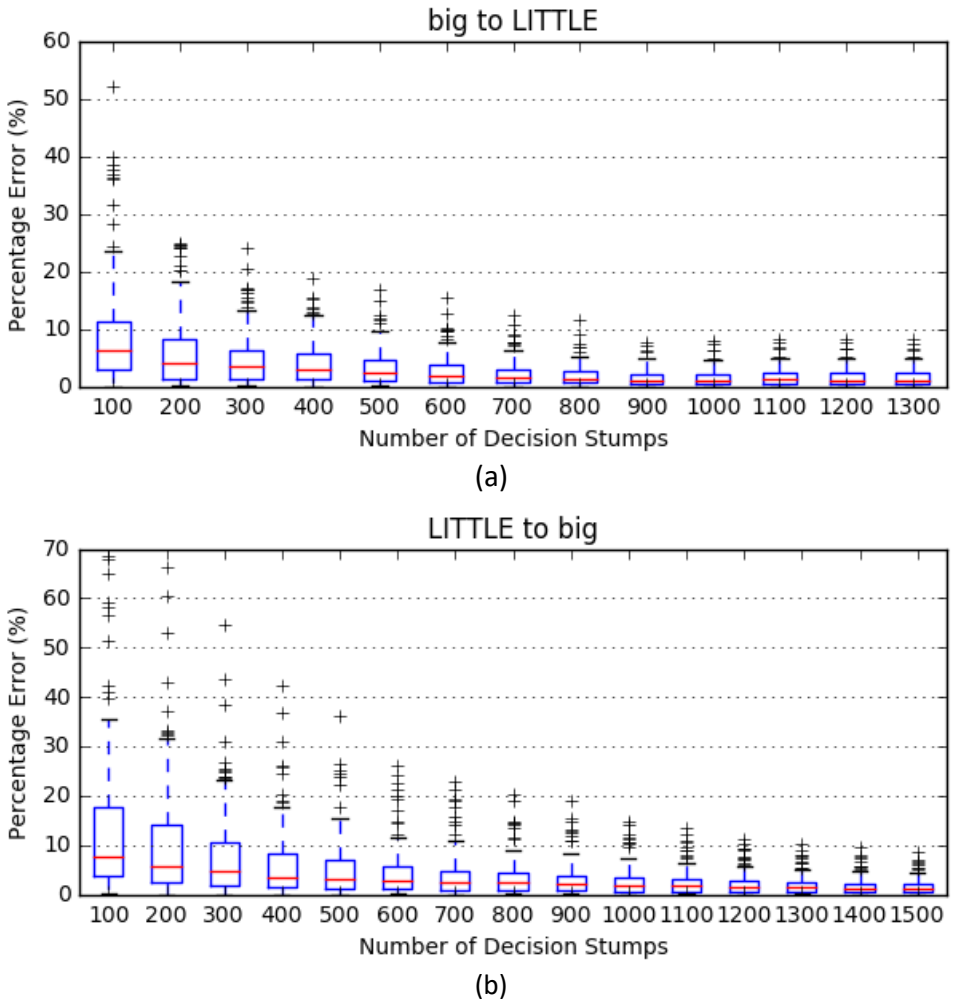

FIGURE 5.4: Box plot of absolute percentage error in IPC prediction by proposed performance model for different number of decision stumps used in the additive regression, showing the median, lower quartile, upper quartile and outliers – (a) Estimating the performance of LITTLE given the information about the big core (b) Estimating the performance of big given the information about the LITTLE core.

Relevant Existing Approaches for Comparison

To show the benefits of proposed approach referred it to as adaptive mapping and DVFS (AdaMD) compared to the state-of-the-art, the selected comparison candidates from the relevant reported works are given below.

1. *HMPx [169]:* The state-of-the-art solution for big.LITTLE multi-processing, the Heterogeneous Multi-Processing (HMP) scheduler, with various default Linux power governors x (= Ondemand (O), Conservative (C) and Interactive (I)) is considered. We did not consider performance and powersave governors as they do not adapt to workload variation and, therefore, constantly operate at the highest and the lowest possible frequency respectively. HMP is a patch to the standard scheduler in the Linux kernel for symmetric multi-processing which dynamically dispatches threads

to big and LITTLE clusters according to their CPU load. For a fair comparison, applications are executed with different numbers of threads and considered the one meeting the performance constraint.

2. *MIM [10]:* This approach maps application threads onto only one type of core(s) based on workload memory-intensity, called a memory-intensity based mapping (MIM). For the single-application execution scenario, a memory-intensive application is mapped onto LITTLE cores, whereas a compute-intensive one is executed on the big cores. In a multiple-application scenario, applications are sorted based on their memory-intensity, and the one with the highest memory-intensity is mapped onto LITTLE cores, and remaining applications are allocated onto the big cluster with an equal number of cores.

3. *EAM [50]:* An energy-efficient mapping is selected through an exhaustive search of voltage-frequency settings and thread-to-core mappings. For each possible thread-to-core mapping, voltage-frequency settings are varied from the lowest possible value to the highest and the one that meets performance requirement with the lowest energy consumption is chosen. We refer to this approach as energy-aware mapping (EAM).

4. *ITMD [57]:* This approach, presented in Chapter 4, uses offline analysis of energy and performance for individual applications to decide on an energy-efficient mapping when multiple applications are run concurrently. Furthermore, it also applies workload classification-based DVFS periodically to minimize the power consumption.

5.3.2 Evaluation of Performance Predictor

The performance prediction model estimates the performance of the big core given the performance of a LITTLE core (P_{bl}) and vice versa (P_{lb}). The number of base learners (decision stumps) M in Eq. 5.3 impacts the model accuracy and runtime overhead. We tested our model over 148 distinct samples to evaluate the model accuracy in IPC estimation and the corresponding box plot of percentage error distribution for P_{bl} and P_{lb} are given in Figures 5.4a and 5.4b respectively. As shown, the error range gets narrower with the number of decision stumps, as it would help in better predicting the speedup. Furthermore, increasing the number of decision stumps also reduces the outliers, shown as cross in Figures 5.4a and 5.4b, improving model stability. However, choosing more decision stumps could increase the runtime overhead, and sometimes accuracy of the prediction may not be improved after reaching a certain number of decision stumps. Therefore, to balance this, additive regression models for different numbers of decision stumps are built. It can be seen from Fig. 5.4a and 5.4b that the the improvement in model accuracy is negligible after 900 and 1100 decision stumps for

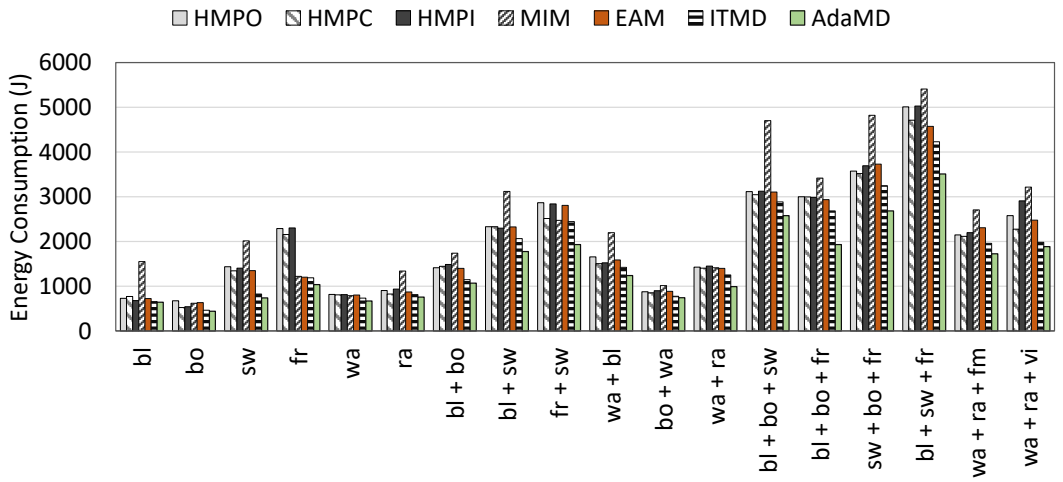

FIGURE 5.5: Comparison of the proposed approach with reported approaches in terms of energy consumption for single and concurrent applications.

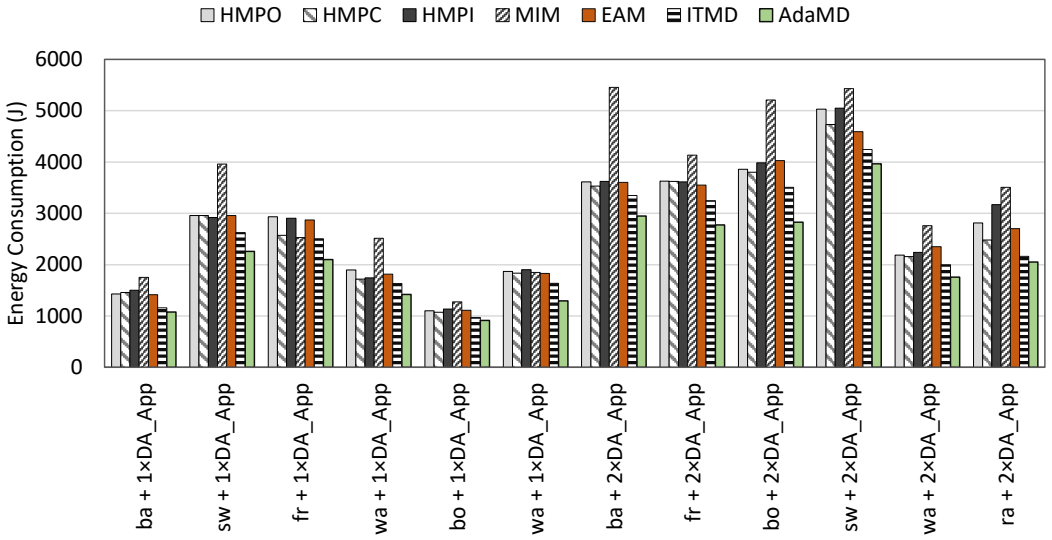

FIGURE 5.6: Energy consumption of different approaches for one and two applications added dynamically to the system while an application is executing.

P_{bl} and P_{lb} respectively. Therefore, these numbers for our models P_{bl} (mean absolute percentage error (MAPE) = 1.57%; maximum error (ME) = 8.1%) and P_{lb} (MAPE = 3.45%; ME = 8.5%) are chosen. The maximum error of P_{bl} and P_{lb} is about 7.9% and 5% lower compared to the previous model [19], respectively. The prediction accuracy of P_{lb} is 1.88% worse than P_{lb} and requires 200 extra decision stumps. This is because the LITTLE cores support accessing only four PMCs simultaneously, compared to six PMCs supported by big cores.

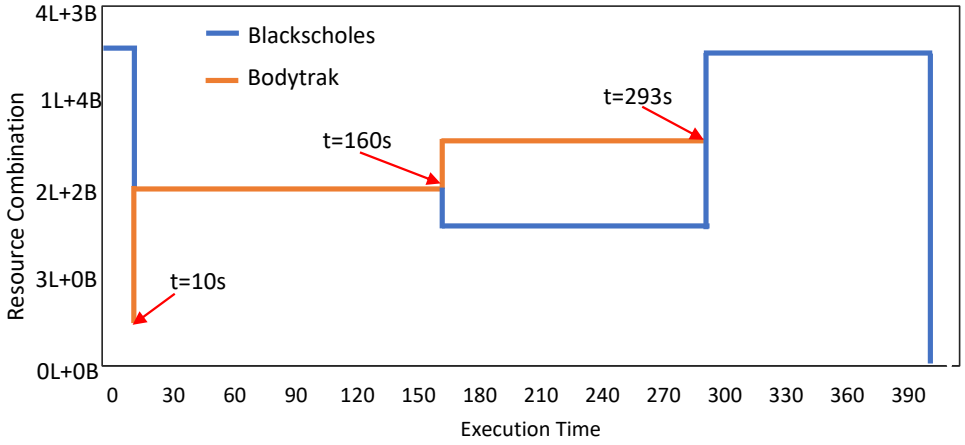

FIGURE 5.7: Resource combination (number of big (B) and LITTLE (L) cores) allocated to `Blackscholes` and `Bodytrack` by the proposed approach to adapt to application arrival/completion and performance variation.

5.3.3 Comparison of Energy Consumption

This section presents the energy consumption results for various approaches to show the benefits of the proposed approach. Fig. 5.5 shows the energy consumption for different single and concurrently executing applications (launched at the same time). We observed substantial energy savings compared to the reported approaches for all the application execution scenarios. For single application execution (bl, bo, sw, fr, wa, and ra), with AdaMD's approach, average energy savings of 30.7%, 25.8%, 27.3%, 37.4%, 21.8% and 7.8% are observed compared to *HMPO*, *HMPC*, *HMPI*, *MIM*, *EAM*, and *ITMD*, respectively. Furthermore, for concurrent execution of two and three applications: bl+bo, bl+sw, fr+sw, wa+bl, bo+wa, wa+ra, bl+bo+sw, bl+bo+fr, sw+bo+fr, bl+sw+fr, wa+ra+fm, wa+ra+vi; *AdaMD* shows 25.5%, 22.4%, 26.5%, 37.5%, 24.8%, and 14.2% lower energy consumption than *HMPO*, *HMPC*, *HMPI*, *MIM*, *EAM*, and *ITMD*, respectively.

In the single application scenario, *ITMD*, *EAM*, and *AdaMD* chooses a similar thread-to-core mapping, however, the energy savings observed are mainly because of the proposed DVFS technique. This can be specifically noticed for applications like bl and bo, which have thread synchronisation overheads of 49% and 62%, which is substantially large compared to the application fr, having less than 4% (with respect to application execution time) [181]. The experimental observations shows that the applications with high synchronisation overheads have lower MRPI values, which would have been wrongly understood as application doing useful heavy computation if bin-classification with utilisation as a second-layer is not considered (please refer to 5.2.5.3). Unlike, *ITMD* and *EAM*, *AdaMD* takes the thread synchronisation overhead into account while selecting a voltage-frequency setting. In concurrent execution scenarios, the energy savings are due to both DVFS and the utilisation of freed resources of a finished application for active applications.

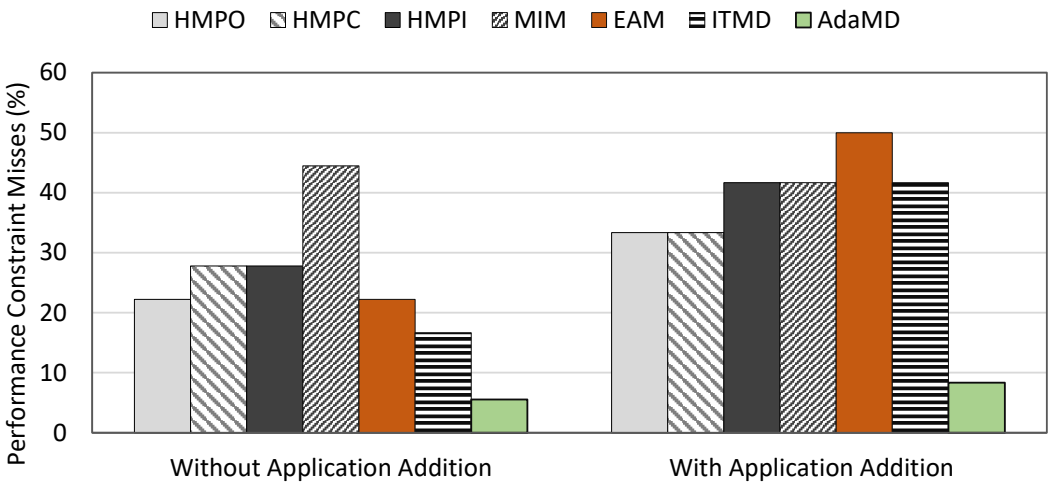

FIGURE 5.8: Evaluation of various approaches in meeting application performance constraints.

Furthermore, to demonstrate the adaptiveness of *AdaMD* to application arrival, the following experimental evaluation is performed. The execution starts with one application and later, one (+ 1×`DA_Apps`) or two (+ 2×`DA_Apps`) applications are added at runtime. The dynamically added applications, abbreviated as DA_Apps in Fig. 5.6, are from those mentioned in Section 5.3.1. The advantages of the proposed approach with respect to other approaches in terms of energy savings are shown in Fig. 5.6. On an average, *AdaMD* reduces the energy consumption by 23.8%, 20.6%, 24.8%, 35.8%, 12.2%, and 23.0% compared to *HMPO*, *HMPC*, *HMPI*, *MIM*, *EAM*, and *ITMD*, respectively. To illustrate how the proposed approach adapts to different runtime scenarios and achieves energy savings, resource combination (number of active core and their type) over execution time for Blackscholes and Bodytrack is shown in Fig. 5.7. While Blackscholes is executing with four LITTLE and three big cores (4L+3B), Bodytrack is added to the system at t=10s. Considering the performance constraints of Bodytrack, 2B+2L are allocated to Bodytrack by freeing the cores from over-performance of Blackscholes. Due to the workload variations, Bodytrack experiences performance loss at t=160s, thereby triggering the Resource Reallocator to readjust the mappings of both Blackscholes and Bodytrack. Upon Bodytrack's completion at t=293s, the freed cores are again allocated to Blackscholes as it can benefit from faster execution to lower the energy consumption. This shows that AdaMD adapts to runtime execution scenarios efficiently and exploits available system resources in an intelligent manner to improve energy efficiency and to satisfy performance constraints.

5.3.4 Performance

The proposed approach outperforms all reported approaches in meeting application performance constraints, as shown in Fig. 5.8. We evaluated the percentage of performance

constraint misses for all the application scenarios presented in Fig. 5.5 (Without Application Addition) and Fig. 5.6 (With Application Addition). For the without application addition case, *AdaMD* meets application performance constraint for 95% of the considered application scenarios, i.e., 17 out of 18 cases, shown on the horizontal axis in Fig. 5.5. The only scenario where the proposed approach fails to satisfy the performance requirements is `bl-sw-fr`. Even in this case, except for `sw`, the performance constraints of `bl` and `fr` are met. This is mainly because of the diverse workload profiles of the three applications and relatively higher performance requirements chosen for `sw` (approximately 2× compared to `bl` and `fr`). In case of application addition at runtime, *AdaMD* is able to satisfy performance constraints for 92% of the evaluated scenarios, i.e., out of 12 scenarios shown in the Fig. 5.6, except for `sw` + 2×`DA_Apps`, the performance constraints are met.

5.3.5 Runtime Overheads

To compute the runtime overheads of the proposed approach, the amount of time that the algorithm takes to complete various steps (A to E), explained in Section 5.2, has been computed. Steps A (Runtime Data Collector), B (Performance Predictor), C (Resource Combination Enumerator) and D (Resource Selector) are triggered when an application arrives into the system, whereas step E (Resource Manager) operates periodically. The runtime overheads can be analytically represented as follows for each time epoch (500 ms):

$$T_o = T_{AdaMap} + 2.5 \times T_{DVFS} \tag{5.7}$$

where,

$$T_{AdaMap} = T_{pmc_m} + T_{pm} + T_{t_h} + T_{rar} \tag{5.8}$$

$$T_{DVFS} = T_{pmc_{vf}} + T_{metrics} + T_{wp} + T_{classify} + T_{vfs} \tag{5.9}$$

where, T_{pmc_m}, T_{pm}, T_{t_h}, T_{rar}, $T_{pmc_{vf}}$, $T_{metrics}$, T_{wp}, $T_{classify}$, and T_{vfs} represent time taken for PMC data collection for mapping; performance prediction; identification of energy-efficient resource combination; resource allocation/reallocation; PMC data collection for DVFS; computation of MRPI and utilisation; workload prediction; finding DVFS setting through classification bins; and DVFS transition latency, respectively. Note that performance prediction happens only when a new application is launched,

therefore the overhead T_{pm} may not be present every time epoch. Moreover, the runtime overhead T_{DVFS} is multiplied with a factor of 2.5×, as it is operated at a finer granularity of 200 ms compared to the mapping time interval of 500 ms.

We observed an average runtime overhead of 600 µs and 1.4 ms for A to D when executed at 2 GHz and 1 GHz on a big core of Odroid-XU3, respectively. The DVFS part of step E incurs 320 µs and others parts take up to 15 µs when the overhead is measured at the maximum frequency (2 GHz). The DVFS algorithm operates at a granularity of 100 ms, so the overhead is less than 0.5%. Performance and Resource manager part of E is invoked for every 500 ms. The overhead associated with this part depends on the number of times the application misses its performance constraint and thread migrations across the cores. Here, an overhead between 0.15% to 0.75% is observed. Our results show that the total runtime overhead is very minimal and moreover, they have been included when computing energy consumption and performance.

5.4 Discussion

The increasing demand for performance and energy efficiency has forced mobile systems to employ heterogeneous multiprocessor system-on-chips. These systems offer a diverse set of core and frequency configurations to runtime management systems for online tuning. This chapter has presented an energy-efficient adaptive thread-to-core mapping and DVFS technique for choosing an appropriate configuration for each performance-constrained application. By using runtime information while applications are executing and eliminating the need for application-dependent offline results, presented approach is capable of managing even unknown applications efficiently. Proposed algorithm first selects a resource combination (number of cores and their type) that meets the application performance requirement using an accurate performance prediction model and resource enumerator/selector. It then monitors application performance, workload and its status (finished or newly arrived) for tuning voltage-frequency settings and adjusting thread-to-core mappings. Our experiments show an improvement of up to 28% in energy efficiency compared to the most promising existing approaches. The proposed approach also outperforms previous approaches in meeting application performance constraints.

Chapter 6

Conclusions and Future Work

This chapter presents the conclusions and closes with the potential future research directions.

6.1 Conclusions

Modern MPSoC platforms with higher number cores of various types have become prevalent due to ever-increasing user demands in terms of performance and energy efficiency. These platforms offer diverse levels of computation such as heterogeneous computing and multi-tasking, and many platform configurations, including runtime adjustment of DVFS settings and thread-to-core mappings. However, due to limited energy supply and the need for longer battery life in the present and future heterogeneous multi-core platforms, runtime energy management has been important. The large platform configurations coupled with concurrent execution of performance-constrained applications has made runtime management a challenging task.

The literature review carried out in Chapter 2 discussed the existing runtime energy management approaches in detail and helped to establish that they have the following shortcomings in handling concurrent execution and dynamic arrival/completion of applications. The previous approaches do not consider the concurrent workload variations and application interference, while selecting DVFS settings, leading to lower energy savings and higher performance degradation. For heterogeneous MPSoCs, existing thread-to-core mapping techniques do not exploit different types of cores simultaneously for each application, which further reduces the total energy savings. Furthermore, it was also established that state-of-the-art approaches targeting heterogeneous MPSoCs, running concurrent applications, are highly dependant on application-specific profile data, and do not perform adaptations at application arrival/completion times. To address the above problems, this thesis made various contributions and their conclusions are presented below.

In Chapter 3, to efficiently choose DVFS settings on a homogeneous MPSoCs for concurrent execution scenarios, an investigation on workload classification based on execution time and statistical analysis of hardware PMC data are presented. Variation in performance of an application, when running with other applications concurrently, is analysed considering various diverse benchmark applications and execution scenarios (sequential and concurrent). This has shown that, unlike compute-intensive workloads, memory-intensive workloads in sequential execution experience small performance degradation when the frequency is scaled down. However, in the case of concurrent execution, the workload profile of individual applications is different than that of sequential execution, thereby making DVFS setting decisions challenging. To solve this, a thorough analysis of runtime statistics collected through hardware PMCs is carried out and finally, a low runtime overhead and efficient metric, called Memory Reads Per Instruction (MRPI) is derived for classifying the concurrent workload. Furthermore, an online energy optimization technique using MRPI is presented for efficient DVFS control. The proposed technique performs an on-the-fly workload classification and pro-actively selects an appropriate DVFS level for a predicted workload. Subsequently, it monitors the workload prediction error and performance loss, quantified by Instructions Per Second (IPS) and adjusts the chosen DVFS setting to compensate. Validation on a homogeneous MPSoC platform (the A15-cluster of the Odroid-XU3) for various execution scenarios showed an improvement of up to 69% in energy consumption compared to existing approaches while incurring a performance loss of up to 4.85%.

This has been then extended to multi-threaded applications involving thread synchronisation overheads, and different DVFS and memory architectures. The extended runtime energy management technique takes the combined effect of application compute-/memory-intensity, thread synchronization contention, and non-uniform memory accesses (NUMAs), for controlling the DVFS setting. This approach showed that accurate estimation of the processor workload periodically is important for efficient DVFS control. In the case of concurrent execution, it has been demonstrated that a careful selection of workload and taking the memory contention into account while selecting a DVFS setting improve energy efficiency with a minimal performance penalty. Experimental results showed an improvement of up to 81.2% in energy efficiency with a small performance loss when compared to existing approaches.

Further, considering the importance of heterogeneous multi-cores executing multi-threaded applications concurrently, an inter-cluster thread-to-core mapping and cluster-based DVFS approach has been presented in Chapter 4. Our investigation shows that mapping an application simultaneously on big and LITTLE cluster improves energy efficiency and performance. For thread-to-core mapping, offline profiled results are used, which contain performance and energy characteristics of applications when executed on the heterogeneous platform at the maximum frequency with different types of cores in various possible combinations. For an application, thread-to-core mapping process defines the number

of used cores and their type, which are situated in different clusters. At runtime, the mapping process identifies thread-to-core mappings for each performance-constrained application in a concurrent execution scenario while optimizing the overall energy consumption. The online adaptation process classifies the workload of concurrently executing applications and pro-actively selects an appropriate V-f pair for a predicted workload. This chapter shows that energy efficiency and performance can be improved by considering the application interference, concurrent workload classification, and inter-cluster thread-to-core-mapping. This approach is also validated on a hardware platform, the Odroid-XU3, with various combinations of multi-threaded applications from PARSEC and SPLASH benchmarks. Results show improvement in energy efficiency of up to 33% compared to existing approaches while meeting the performance requirements.

To remove the dependency on the application-dependant profile data and offline instrumentation of applications, and to adapt to application arrival and finish times, Chapter 5 has presented an energy-efficient adaptive thread-to-core mapping and DVFS technique for choosing an appropriate configuration for each performance-constrained application. By using runtime information while applications are executing and eliminating the dependency on the application-dependent offline results, this approach is capable of managing even unknown applications efficiently. Proposed approach first selects a resource combination (number of cores and their type) that meets the application performance requirement using an accurate performance prediction model and resource enumerator/selector. It then monitors application performance, workload and its status (finished or newly arrived) for tuning voltage-frequency settings and adjusting thread-to-core mappings. Our experiments show an improvement of up to 28% in energy efficiency compared to the most promising existing approaches. The proposed approach also outperforms previous approaches in meeting application performance constraints.

The future computing platforms such as embedded/mobile are expected to have an increased number of processing cores of various types to meet diverse energy and performance demands of applications. However, architectural innovations alone may not fully exploit the application characteristics to achieve energy efficiency while respecting the performance constraints. Therefore, as reported in this thesis, intelligent runtime managers would complement architectural innovations to exploit hardware and applications characteristics efficiently, thereby improving various metrics such as energy, performance, etc. In this context, the advances reported in this thesis are significant contributions towards the development of future energy efficient, feature-rich multi-core platforms. The following section outlines the future research directions that are worth considering to improve certain features of the presented contributions and also identifies areas which may benefit from this thesis investigations.

6.2 Future Work

This section highlights some potential future research directions to extend or augment the work presented in this thesis.

1. *Validation with Interactive Mobile Workloads:* The presented RTM approaches have been validated with several standard benchmarks to show their advantages in terms of energy efficiency and performance. The next interesting step would be to validate with mobile applications (e.g., Web browser, Facebook, Twitter, Instagram, etc.) that generate interactive workloads depending upon the interplay between processing cores, system peripherals and the user, such as touch screen usage during web-browsing. Due to time constraints, this has not been carried out during the course of PhD. Such validation may help identify and tune certain features of the RTM to manage interactive mobile workloads efficiently.

2. *Joint Optimization of Mapping and DVFS:* To make runtime decisions with negligible overhead, the RTM approaches presented in Chapter 4 and 5 consider mapping and DVFS separately. However, this may result in a selection of sub-optimal platform configurations due to limited design space, leading to reduced energy savings and/or performance. Therefore, investigating the benefits of joint optimization of mapping and DVFS would be a good extension to this thesis contributions. It would certainly require lower overhead and efficient techniques to make such runtime optimization beneficial. Furthermore, dynamic power management to gracefully shutdown unused components in a system would also be good to investigate along with the above.

3. *Variable Sample Period for DVFS:* The time period considered for detecting workload variations and adjusting the DVFS setting has been constant in this thesis. However, some applications may exhibit workload variations quite frequently than others, which would also depend upon the interaction between the application and the user. Therefore, using a uniform and constant time period may not be effective in such scenarios. Therefore, investigating methods to tune the DVFS sample period may lead to better capture workload variations and thus reducing power consumption.

4. *Integration of Runtime Manager into Scheduler:* Recent research on energy-aware scheduling has shown promising results in terms of improved energy efficiency and performance. Therefore, the approaches presented in this thesis and above future research work on runtime managers can be integrated into scheduler to identify possible benefits and to improve the system performance at large.

5. *Adaptive Energy Management for Massively Heterogeneous Systems:* The prior art showed that applications from different areas (e.g., data mining, multimedia,

machine learning, etc.) exhibit distinct workload characteristics. Executing such applications on a less sophisticated hardware platform may not exploit the workload characteristics efficiently, thereby leading to higher energy consumption whilst achieving the same or lower performance than a massively heterogeneous system. We envisage that a massively parallel system contains domain-specific system-on-chips (DS-SoC), reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs), specialized hardware accelerators like Tensor Processing Units (TPUs), Graphics Processing units (GPUs), etc. This kind of complex systems necessitate a highly sophisticated runtime management systems that are scalable in terms of processing cores heterogeneity; adaptable to various runtime variations, including performance and workload. The presented ideas for runtime task mapping and DVFS can be extended to such systems.

6. *A holistic runtime management approach to improving performance, power, temperature, security and reliability:* in addition to performance, power and temperature, metrics such as security and reliability have gained significant importance mainly due to highly connected devices posing hardware and software security threats, technology scaling, and frequent device failures. Optimizing a particular metric may hurt the other, for example, minimizing power consumption and device temperature without considering the performance may lead to under-utilization of platform resources. Therefore, current and future runtime management systems should consider a holistic approach to have a better understanding of the relationship between metrics and design an efficient manager. This would lead to meeting the power, performance, or temperature constraints while providing reasonable device security and performance guarantees event under device errors/failures.

7. *Platform- and Application-Agnostic Runtime Management System:* There have been continuous advancements in architecture, process technology, and the emergence of new applications. A runtime management software targeting a particular application, domain, or hardware platform may become obsolete or inefficient due to rapid advancements. Therefore, ideas presented in Chapter 5 can be extended to incorporate platform-agnostic runtime management by deriving generalized metrics (mostly software) for quantifying various runtime objectives, such as power, performance, reliability, etc. This necessitates a thorough understanding of state-of-the-art architectural and application innovations, and trends in future advancements. As a result, such a generalized runtime manager can be highly scalable, and if necessary, it can be tuned to suit a particular hardware architecture or application domain to exploit certain features to improve, for example, energy efficiency.

Appendix A

Application Offline Analysis Results

This appendix contains the application offline profiling results used in Chapter 4 for making thread-to-core mapping decisions.

TABLE A.1: Offline profiling results for **Blackscholes**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	1328.65	2058.99
	0L+1B	883.55	2327.26
2	2L	1274.90	1993.47
	1L+1B	885.62	2359.80
	0L+2B	880.62	2325.78
3	3L	722.72	1206.00
	2L+1B	888.42	2351.88
	1L+2B	476.50	1812.30
	0L+3B	473.69	1799.32
4	4L	546.48	949.58
	3L+1B	881.45	2352.01
	2L+2B	450.12	1652.92
	1L+3B	351.39	1550.33
	0L+4B	353.95	1582.22
5	4L+1B	882.14	2351.62
	3L+2B	496.11	1832.64
	2L+3B	327.25	1323.90
	1L+4B	320.39	1297.37
6	4L+2B	462.08	1759.06
	3L+3B	278.25	1173.21
	2L+4B	290.52	1057.66
7	4L+3B	257.12	1028.13
	3L+4B	246.33	918.16
8	4L+4B	226.13	885.19

TABLE A.2: Offline profiling results for **Bodytrack**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	1164.18	1797.35
	0L+1B	402.83	1017.32
2	2L	1128.68	1728.67
	1L+1B	418.45	1039.23
	0L+2B	399.11	1024.35
3	3L	1110.10	1706.33
	2L+1B	400.62	1022.49
	1L+2B	412.83	1020.65
	0L+3B	418.14	1058.84
4	4L	589.53	955.45
	3L+1B	412.02	1058.96
	2L+2B	228.58	765.54
	1L+3B	234.68	769.55
	0L+4B	231.01	773.02
5	4L+1B	408.50	1044.24
	3L+2B	240.60	779.77
	2L+3B	183.19	653.02
	1L+4B	182.42	657.29
6	4L+2B	236.85	760.61
	3L+3B	181.60	644.10
	2L+4B	171.17	592.20
7	4L+3B	178.35	633.86
	3L+4B	159.80	577.97
8	4L+4B	152.89	500.91

TABLE A.3: Offline profiling results for **Swaptions**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	2181.22	3445.20
	0L+1B	1037.45	2510.84
2	2L	2099.98	3277.43
	1L+1B	1038.14	2436.14
	0L+2B	1039.45	2474.41
3	3L	1052.34	1644.25
	2L+1B	1039.94	2434.03
	1L+2B	522.94	1160.14
	0L+3B	530.84	1218.90
4	4L	727.01	1139.44
	3L+1B	1037.07	2310.05
	2L+2B	524.56	1191.67
	1L+3B	410.74	955.67
	0L+4B	409.30	955.75
5	4L+1B	1039.14	2358.40
	3L+2B	521.46	1398.28
	2L+3B	419.96	1105.55
	1L+4B	352.80	913.61
6	4L+2B	521.35	1375.12
	3L+3B	345.05	991.88
	2L+4B	351.13	950.46
7	4L+3B	311.05	849.38
	3L+4B	297.33	797.46
8	4L+4B	282.99	715.22

TABLE A.4: Offline profiling results for **Freqmine**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	2790.53	4514.70
	0L+1B	2775.68	8782.74
2	2L	1415.05	2296.33
	1L+1B	894.05	2821.96
	0L+2B	895.30	2791.99
3	3L	967.57	1565.99
	2L+1B	898.87	2844.26
	1L+2B	553.52	1734.41
	0L+3B	568.56	1773.00
4	4L	763.51	1376.72
	3L+1B	887.17	2799.31
	2L+2B	552.28	2646.77
	1L+3B	477.13	2285.95
	0L+4B	474.89	2299.49
5	4L+1B	897.95	2818.19
	3L+2B	536.63	2573.14
	2L+3B	432.25	2075.05
	1L+4B	425.45	2046.62
6	4L+2B	484.96	2165.45
	3L+3B	387.11	2149.71
	2L+4B	412.91	2297.40
7	4L+3B	411.70	2225.60
	3L+4B	357.59	1934.66
8	4L+4B	378.41	2045.00

TABLE A.5: Offline profiling results for **Streamcluster**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	2588.80	4137.59
	0L+1B	881.84	2694.36
2	2L	2476.25	3935.27
	1L+1B	883.20	2688.13
	0L+2B	886.76	2679.46
3	3L	2463.82	4424.74
	2L+1B	882.30	2701.43
	1L+2B	882.14	4545.92
	0L+3B	885.90	4449.48
4	4L	1282.76	2560.50
	3L+1B	908.47	2774.88
	2L+2B	515.23	2406.14
	1L+3B	514.67	2765.96
	0L+4B	515.00	2812.07
5	4L+1B	907.77	2771.59
	3L+2B	516.11	2662.74
	2L+3B	515.42	2475.65
	1L+4B	515.28	2732.45
6	4L+2B	587.31	3111.27
	3L+3B	397.75	2074.12
	2L+4B	399.47	1948.54
7	4L+3B	403.40	2140.17
	3L+4B	402.23	2154.34
8	4L+4B	365.15	1891.56

TABLE A.6: Offline profiling results for **Vips**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	993.70	1586.06
	0L+1B	415.60	680.93
2	2L	949.52	1732.09
	1L+1B	445.18	1403.84
	0L+2B	405.96	1273.09
3	3L	945.88	1885.60
	2L+1B	440.75	1387.54
	1L+2B	254.09	1264.97
	0L+3B	462.34	2189.35
4	4L	1024.71	2208.15
	3L+1B	273.31	864.04
	2L+2B	460.12	2267.79
	1L+3B	408.13	1980.98
	0L+4B	404.36	2050.51
5	4L+1B	469.31	1477.87
	3L+2B	283.94	1420.65
	2L+3B	283.97	1389.51
	1L+4B	333.75	1568.48
6	4L+2B	295.91	1499.53
	3L+3B	243.39	1194.39
	2L+4B	306.86	1397.03
7	4L+3B	258.11	1175.81
	3L+4B	244.99	1177.07
8	4L+4B	301.73	1305.84

TABLE A.7: Offline profiling results for **Water_Spatial**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	1787.21	2812.24
	2L	812.59	1286.46
2	0L+1B	689.48	1788.34
	3L	572.58	882.37
	4L+1B	528.30	1373.47
3	3L+1B	504.37	1308.34
	1L+1B	493.16	1306.41
	2L+1B	447.56	1155.18
	4L	429.63	646.38
4	0L+2B	339.60	914.92
	1L+2B	334.17	894.93
	3L+2B	309.61	818.89
	4L+2B	309.54	808.20
	2L+2B	303.57	796.57
5	0L+3B	265.76	711.39
	1L+3B	258.72	697.55
	1L+4B	226.84	606.37
	2L+3B	225.10	593.21
6	0L+4B	224.08	589.21
	2L+4B	207.33	546.34
	3L+3B	205.45	522.74
7	4L+3B	191.02	506.21
	3L+4B	186.06	496.19
8	4L+4B	175.53	451.55

TABLE A.8: Offline profiling results for **Raytrace**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	2343.84	3717.67
	2L	1290.41	2250.87
2	3L	915.93	1756.57
	4L	732.71	1512.31
	4L+1B	498.88	1548.78
3	3L+1B	488.07	1529.05
	2L+1B	473.71	1521.97
	1L+1B	457.71	1432.97
	0L+1B	444.58	1296.94
4	3L+2B	278.44	1374.51
	4L+2B	276.70	1383.30
	2L+2B	270.33	1333.19
	1L+2B	258.37	1204.50
	0L+2B	251.70	1180.44
5	0L+3B	226.89	1140.63
	1L+3B	207.02	999.05
	0L+4B	201.31	992.15
	2L+3B	192.08	969.55
6	3L+3B	191.43	969.39
	1L+4B	191.23	897.53
	4L+3B	184.48	933.73
7	2L+4B	179.64	867.52
	4L+4B	178.74	898.58
8	3L+4B	171.47	858.35

TABLE A.9: Offline profiling results for **Fmm**, executing on the Odroid-XU3

# cores	Resource combination	Execution Time (s)	Energy Consumption (J)
1	1L	14.48	22.81
	4L	13.82	27.36
2	3L	13.79	25.73
	2L	13.79	23.63
	0L+1B	9.57	31.61
3	3L+2B	9.55	48.19
	3L+1B	9.54	30.57
	2L+1B	9.54	30.78
	4L+1B	9.54	30.27
4	1L+1B	9.53	31.13
	4L+2B	9.52	49.65
	4L+3B	9.52	51.30
	2L+3B	9.51	53.18
	2L+2B	9.51	48.75
5	1L+4B	9.51	51.69
	4L+4B	9.51	52.14
	2L+4B	9.50	51.67
	0L+3B	9.50	52.57
6	0L+4B	9.50	51.34
	3L+3B	9.50	52.99
	0L+2B	9.49	49.89
7	1L+2B	9.49	49.51
	1L+3B	9.49	52.90
8	3L+4B	9.48	51.52

Appendix B

Experimental Data Used in Classification Bins

This appendix contains the experimental data used in DVFS governors, presented in Section 3.3 of Chapter 3 and Section 5.2.5.3 of Chapter 5.

TABLE B.1: The classification bins, namely - utilisation and MRPI, and corresponding frequencies for odroid-XU3

Big			**LITTLE**		
Utilisation	MRPI	Frequency (×100 MHz)	Utilisation	MRPI	Frequency (×100 MHz)
>95.0	>0.02	8	>95.0	>0.04	8
	(0.01, 0.02]	9		(0.03, 0.04]	9
	(0.009, 0.01]	10		(0.02, 0.03]	10
	(0.008, 0.009]	11		(0.01, 0.02]	11
	(0.007, 0.008]	12		(0.095, 0.01]	12
	(0.006, 0.007]	13		(0.009, 0.0095]	13
	(0.005, 0.006]	14		<=0.009	14
	(0.0045, 0.005]	15	(85.0, 95.0]	>0.03	8
	(0.004, 0.0045]	16		(0.02, 0.03]	9
	(0.0035, 0.004]	17		(0.01, 0.02]	10
	(0.003, 0.0035]	18		(0.095, 0.01]	11
	(0.0025, 0.003]	19		(0.009, 0.0095]	12
	<=0.0025	20		<=0.009	13
(85.0, 95.0]	>0.01	8	(75.0, 85.0]	>0.02	9
	(0.009, 0.01]	9		(0.01, 0.02]	10
	(0.008, 0.009]	10		(0.095, 0.01]	11
	(0.007, 0.008]	11		(0.009, 0.0095]	12
	(0.006, 0.007]	12		<=0.009	13
	(0.005, 0.006]	13	(65.0, 75.0]	>0.01	8
	(0.0045, 0.005]	14		(0.095, 0.01]	9
	(0.004, 0.0045]	15		(0.009, 0.0095]	10
	(0.0035, 0.004]	16		<=0.009	11
	(0.003, 0.0035]	17	(55.0, 65.0]	>0.0095	7
	<=0.003	18		(0.009, 0.0095]	8

TABLE B.2: Classifications bins data continued from Table B.1

Big			**LITTLE**		
Utilisation	MRPI	Frequency (x100 MHz)	Utilisation	MRPI	Frequency (x100 MHz)
(75.0, 85.0]	>0.009	9	(55.0, 65.0]	(0.008, 0.009]	9
	(0.008, 0.009]	10		<=0.008	10
	(0.007, 0.008]	11	(45.0, 55.0]	>0.009	6
	(0.006, 0.007]	12		(0.008, 0.009]	7
	(0.005, 0.006]	13		(0.007, 0.008]	8
	(0.0045, 0.005]	14		<= 0.007	9
	(0.004, 0.0045]	15	(35.0, 45.0]	>0.008	5
	(0.0035, 0.004]	16		(0.007, 0.008]	6
	<=0.0035	17		<= 0.007	7
(65.0, 75.0]	>0.008	8	(25.0, 35.0]	>0.007	4
(55.0, 65.0]	(0.007, 0.008]	9		<= 0.007	3
	(0.006, 0.007]	10	<=15.0	-	2
	(0.005, 0.006]	11			
	(0.0045, 0.005]	12			
	(0.004, 0.0045]	13			
	(0.0035, 0.004]	14			
	<=0.0035	15			
	>0.007	7			
	(0.006, 0.007]	8			
	(0.005, 0.006]	9			
	(0.0045, 0.005]	10			
	(0.004, 0.0045]	11			
	(0.0035, 0.004]	12			
	<=0.0035	13			
(45.0, 55.0]	>0.006	6			
	(0.005, 0.006]	7			
	(0.0045, 0.005]	8			
	(0.004, 0.0045]	9			
	(0.0035, 0.004]	10			
	<=0.0035	11			
(35.0, 45.0]	>0.005	5			
	(0.0045, 0.005]	6			
	(0.004, 0.0045]	7			
	(0.0035, 0.004]	8			
	<=0.0035	9			
(25.0, 35.0]	>0.0045	4			
	(0.004, 0.0045]	5			
	(0.0035, 0.004]	6			
	<0.0035	7			
(15.0, 25.0]	>0.004	3			
	<=0.004	2			
<15.0	-	2			

TABLE B.3: The classification bins, namely - utilisation and MRPI, and corresponding frequencies for Intel Xeon E5-2630

Utilisation	MRPI	Frequency (x100 MHz)	Utilisation	MRPI	Frequency (x100 MHz)
>95.0	>0.26	17	(55.0, 65.0]	>0.18	14
	(0.24, 0.26]	18		(0.16, 0.18]	15
	(0.22, 0.24]	19		(0.14, 0.16]	16
	(0.20, 0.22]	20		(0.12, 0.14]	17
	(0.18, 0.20]	21		(0.10, 0.12]	18
	(0.16, 0.18]	22		(0.08, 0.10]	19
	(0.14, 0.16]	23		(0.06, 0.08]	20
	(0.12, 0.14]	24		<=0.06	21
	(0.10, 0.12]	25	(45.0, 55.0]	>0.16	13
	<=0.10	26		(0.14, 0.16]	14
(85.0, 95.0]	>0.24	17		(0.12, 0.14]	15
	(0.22, 0.24]	18		(0.10, 0.12]	16
	(0.20, 0.22]	19		(0.08, 0.10]	17
	(0.18, 0.20]	20		(0.06, 0.08]	18
	(0.16, 0.18]	21		(0.04, 0.06]	19
	(0.14, 0.16]	22		<=0.04	20
	(0.12, 0.14]	23	(35.0, 45.0]	>0.14	12
	(0.10, 0.12]	24		(0.12, 0.14]	13
	<=0.10	25		(0.10, 0.12]	14
(75.0, 85.0]	>0.22	16		(0.08, 0.10]	15
	(0.20, 0.22]	17		(0.06, 0.08]	16
	(0.18, 0.20]	18		(0.04, 0.06]	17
	(0.16, 0.18]	19		<=0.04	18
	(0.14, 0.16]	20	(25.0, 35.0]	>0.12	12
	(0.12, 0.14]	21		(0.10, 0.12]	13
	(0.10, 0.12]	22		(0.08, 0.10]	14
	(0.08, 0.10]	23		(0.06, 0.08]	15
	<=0.08	24		<=0.06	16
(65.0, 75.0]	>0.20	15	(15.0, 25.0]	>0.10	12
	(0.18, 0.20]	16		(0.08, 0.10]	13
	(0.16, 0.18]	17		<=0.08	14
	(0.14, 0.16]	18	<15.0	-	12
	(0.12, 0.14]	19			
	(0.10, 0.12]	20			
	(0.08, 0.10]	21			
	(0.06, 0.08]	22			
	<=0.06	23			

Appendix C

Sample Code for Data Collection and Processing

This appendix contains the sample code used to collect and process data collected for experimental validation.

```
############################################################
#Power_Sensors.sh
#Script for Sampling power sensors on the Odroid-XU3
#Based on sample script from https://forum.odroid.com/
Modified by Karunakar R. Basireddy (krb1g15@soton.ac.uk)
############################################################
#!/bin/bash

# enable the sensors
echo 1 > /sys/bus/i2c/drivers/INA231/3-0045/enable
echo 1 > /sys/bus/i2c/drivers/INA231/3-0040/enable
echo 1 > /sys/bus/i2c/drivers/INA231/3-0041/enable
echo 1 > /sys/bus/i2c/drivers/INA231/3-0044/enable

# settle one second to the sensors get fully enabled and have the first reading
sleep 1

# Main infinite loop
while true; do
# ------ CPU DATA ------ #

A7_W=‘cat /sys/bus/i2c/drivers/INA231/3-0045/sensor_W‘
A15_W=‘cat /sys/bus/i2c/drivers/INA231/3-0040/sensor_W‘
```

```
# ------ Memory DATA ------ #

MEM_W=‘cat /sys/bus/i2c/drivers/INA231/3-0041/sensor_W‘

# Writing the data to files #

echo "$A15_W" ».//Results/Power_A15.txt
echo "$A7_W" ».//Results/Power_A7.txt
echo "$MEM_W" ».//Results/Power_Memory.txt
sleep 0.1

#Checking if application has finished the execution #

a=‘cat status.txt‘
if [ $a -eq 1 ]
then
exit
fi
done
```

```
/*#############################################################
/ /* Read the RAPL registers on recent (>sandybridge) Intel processors */
/* MSR Code originally based on a (never made it upstream) linux-kernel */
/* RAPL driver by Zhang Rui <rui.zhang@intel.com> */
/* https://lkml.org/lkml/2011/5/26/93 */
/* Additional contributions by:  Romain Dolbeau - romain @ dolbeau.org /
/* For raw MSR access the /dev/cpu/??/msr driver must be enabled and */
/* permissions set to allow read access.  */
/* You might need to "modprobe msr" before it will work.  */
/* perf_event_open() support requires at least Linux 3.14 and to have */
/* /proc/sys/kernel/perf_event_paranoid < 1 */
/* the sysfs powercap interface got into the kernel in 2d281d8196e38dd (3.13)
*/
/* Compile with:  gcc -O2 -Wall -o rapl-read rapl-read.c -lm */
/* Vince Weaver - vincent.weaver @ maine.edu - 11 September 2015 */
/* */
/* Modified by
Karunakar:  krb1g15@soton.ac.uk
```

```
Date:  9-March-2018
/ /*############################################################*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <inttypes.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <sys/syscall.h>
#include <linux/perf_event.h>
#define MSR_RAPL_POWER_UNIT 0x606
/*
* Platform specific RAPL Domains:  Note that PP1 RAPL Domain is supported
on 062A only and DRAM RAPL Domain is supported on 062D only
*/
/* Package RAPL Domain */
#define MSR_PKG_RAPL_POWER_LIMIT 0x610
#define MSR_PKG_ENERGY_STATUS 0x611
#define MSR_PKG_PERF_STATUS 0x613
#define MSR_PKG_POWER_INFO 0x614

/* PP0 RAPL Domain */
#define MSR_PP0_POWER_LIMIT 0x638
#define MSR_PP0_ENERGY_STATUS 0x639
#define MSR_PP0_POLICY 0x63A
#define MSR_PP0_PERF_STATUS 0x63B
/* PP1 RAPL Domain, may reflect to uncore devices */
#define MSR_PP1_POWER_LIMIT 0x640
#define MSR_PP1_ENERGY_STATUS 0x641
#define MSR_PP1_POLICY 0x642
/* DRAM RAPL Domain */
#define MSR_DRAM_POWER_LIMIT 0x618
#define MSR_DRAM_ENERGY_STATUS 0x619
#define MSR_DRAM_PERF_STATUS 0x61B
#define MSR_DRAM_POWER_INFO 0x61C
```

```
/* PSYS RAPL Domain */
#define MSR_PLATFORM_ENERGY_STATUS 0x64d
/* RAPL UNIT BITMASK */
#define POWER_UNIT_OFFSET 0
#define POWER_UNIT_MASK 0x0F
#define ENERGY_UNIT_OFFSET 0x08
#define ENERGY_UNIT_MASK 0x1F00
#define TIME_UNIT_OFFSET 0x10
#define TIME_UNIT_MASK 0xF000
static int open_msr(int core) {
char msr_filename[BUFSIZ];
int fd;
sprintf(msr_filename, "/dev/cpu/%d/msr", core);
fd = open(msr_filename, O_RDONLY);
if ( fd < 0 ) {
if ( errno == ENXIO ) {
fprintf(stderr, "rdmsr:  No CPU %d\n", core);
exit(2);
} else if ( errno == EIO ) {
fprintf(stderr, "rdmsr:  CPU %d doesn't support MSRs\n", core);
exit(3);
} else {
perror("rdmsr:open");
fprintf(stderr,"Trying to open %s\n",msr_filename);
exit(127);
}
}
return fd;
}
static long long read_msr(int fd, int which) {
uint64_t data;
if ( pread(fd, &data, sizeof data, which) != sizeof data ) {
perror("rdmsr:pread");
exit(127);
}
return (long long)data;
}
#define CPU_SANDYBRIDGE 42
#define CPU_SANDYBRIDGE_EP 45
#define CPU_IVYBRIDGE 58
#define CPU_IVYBRIDGE_EP 62
```

```
#define CPU_HASWELL 60
#define CPU_HASWELL_ULT 69
#define CPU_HASWELL_GT3E 70
#define CPU_HASWELL_EP 63
#define CPU_BROADWELL 61
#define CPU_BROADWELL_GT3E 71
#define CPU_BROADWELL_EP 79
#define CPU_BROADWELL_DE 86
#define CPU_SKYLAKE 78
#define CPU_SKYLAKE_HS 94
#define CPU_SKYLAKE_X 85
#define CPU_KNIGHTS_LANDING 87
#define CPU_KNIGHTS_MILL 133
#define CPU_KABYLAKE_MOBILE 142
#define CPU_KABYLAKE 158
#define CPU_ATOM_SILVERMONT 55
#define CPU_ATOM_AIRMONT 76
#define CPU_ATOM_MERRIFIELD 74
#define CPU_ATOM_MOOREFIELD 90
#define CPU_ATOM_GOLDMONT 92
#define CPU_ATOM_GEMINI_LAKE 122
#define CPU_ATOM_DENVERTON 95
/*Collecting CPU information*/
static int detect_cpu(void) {
FILE *fff;
int family,model=-1;
char buffer[BUFSIZ],*result;
char vendor[BUFSIZ];
fff=fopen("/proc/cpuinfo","r");
if (fff==NULL) return -1;
while(1) { result=fgets(buffer,BUFSIZ,fff); if (result==NULL) break; if (!strncmp(res
{
sscanf(result,"%*s%*s%s",vendor);
if (strncmp(vendor,"GenuineIntel",12)) {
printf("%s not an Intel chip\n",vendor);
return -1;
}
}
if (!strncmp(result,"cpu family",10)) {
sscanf(result,"%*s%*s%*s%d",&family);
if (family!=6) {
```

```
printf("Wrong CPU family %d\n",family);
return -1;
}
}
if (!strncmp(result,"model",5)) {
sscanf(result,"%*s%*s%d",&model);
}
}
fclose(fff);
return model;
} #define MAX_CPUS 1024
#define MAX_PACKAGES 16

static int total_cores=0, total_packages=0;
static int package_map[MAX_PACKAGES];
static int detect_packages(void) {
char filename[BUFSIZ];
FILE *fff;
int package;
int i;
for(i=0;i<MAX_PACKAGES;i++) package_map[i]=-1;
printf("\t");
for(i=0;i<MAX_CPUS;i++) {
sprintf(filename,"/sys/devices/system/cpu/cpu%d/topology/physical_package_id",i);
fff=fopen(filename,"r");
if (fff==NULL) break;
fscanf(fff,"%d",&package);
fclose(fff);
if (package_map[package]==-1) {
total_packages++;
package_map[package]=i;
}
}
total_cores=i;
return 0;
}

/*******************************/
/* MSR code */
/*******************************/
static int rapl_msr(int core, int cpu_model, useconds_t S_period) {
```

```
int fd;
long long result;
double power_units, time_units;
double cpu_energy_units[MAX_PACKAGES], dram_energy_units[MAX_PACKAGES];
double pp0_before[MAX_PACKAGES], pp0_after[MAX_PACKAGES], total_energy;
int j;
int dram_avail=0, pp0_avail=0, pp1_avail=0, psys_avail=0;
int different_units=0;

if (cpu_model<0) {
printf("\tUnsupported CPU model %d\n",cpu_model);
return -1;
}
//Setting Package info based on cpu model - CPU_IVYBRIDGE_EP

pp0_avail=1;
pp1_avail=0;
dram_avail=1;
different_units=0;
psys_avail=0;
break;

for(j=0;j<total_packages;j++) {
fd=open_msr(package_map[j]);

/* Calculate the units used */
result=read_msr(fd,MSR_RAPL_POWER_UNIT);
power_units=pow(0.5,(double)(result&0xf));
cpu_energy_units[j]=pow(0.5,(double)((result>>8)&0x1f));
time_units=pow(0.5,(double)((result>>16)&0xf));

/* The DRAM units differ from the CPU ones */
if (different_units) {
dram_energy_units[j]=pow(0.5,(double)16);
}
else {
dram_energy_units[j]=cpu_energy_units[j];
}
close(fd);
}
```

```
for(j=0;j<total_packages;j++) {
fd=open_msr(package_map[j]);
/* PP0 energy */
if (pp0_avail) {
result=read_msr(fd,MSR_PP0_ENERGY_STATUS);
pp0_before[j]=(double)result*cpu_energy_units[j];
}
close(fd);
}
usleep(S_period);
int monitor=1;
FILE *mon, *pow;
total_energy = 0.0;
pow = fopen("/home/karun/Benchmarks/Xeon/EvalData/Energy_data.csv", "a");
//Checking if application has finished the execution
while(monitor)
{
mon = fopen("/home/karun/Benchmarks/Xeon/EvalData/Monitor_Cntl.txt", "r");
fscanf(mon, "%d", &monitor);
fclose(mon);
for(j=0;j<total_packages;j++) {
fd=open_msr(package_map[j]);
result=read_msr(fd, MSR_PP0_ENERGY_STATUS);
pp0_after[j]=(double)result*cpu_energy_units[j];
total_energy += pp0_after[j] - pp0_before[j];
pp0_before[j] = pp0_after[j];
close(fd);
}
fprintf(pow, "%.6f\n", total_energy);
usleep(S_period);
total_energy = 0.0;
}
fclose (pow);
printf("Tired of sampling and writing to a file; I'm quitting now \n \n");
return 0;
}

int main(int argc, char **argv) {
int core=0;
int result=-1;
int cpu_model;
```

```
int c = 0;
useconds_t S_period = 50000;
opterr = 0;
while ((c = getopt (argc, argv, "t:h")) != -1){
switch (c)
{
case 't':
S_period = atoi(optarg);
break;
case 'h':
default:
printf("Usage:  %s [-h] [-t] sampling_period(us) \t\n", argv[0]);
return 1;
}
}
cpu_model=detect_cpu();
detect_packages();
printf("----------------------------------\n");
printf("Starting power sampling (for every %u micro-seconds)\n", (unsigned
int)S_period);
printf("----------------------------------\n");
result = rapl_msr(core, cpu_model, S_period);
if (result<0) {
printf("Unable to read RAPL counters.\n");
printf("* If using raw msr access, make sure msr module is installed\n");
printf("\n");
return -1;
}
return 0;
}
```

```
############################################################
#Power_Process.sh
#Script for processing power/energy data
Created by Karunakar R. Basireddy (krb1g15@soton.ac.uk)
############################################################
#!/bin/bash

if [ "$#" -ne 2 ]
then
```

```
echo "Usage:  $0 <Approach_name> <App_Scenario>" >&2
exit 1
fi

a=$1
b=$2
#Core=$1

#creates a directory Trimmed_Data if it doesn't exist
DIR="<dir_path>"
if [ -d "$DIR" ]; then
echo -e ""Seems directory exists, it may over write the contents!  exiting"\n
\t"
exit
else
mkdir /home/karunakar/<path>/process-data
exit
fi

DIR_D="/home/karunakar/<path>/process-data"
DATD="/home/karunakar/<path>/b/a"
#Moves the data between each START and FINISHED to a new file

sed 's/«««««START»»»»»»/\t&\n/g;s/^\n\(«««««START»»»»»»\)/\1\n/;\
s/«««««FINISHED»»»»»»/\n&\n/g;s/^\n\(«««««FINISHED»»»»»»\)/\1\n/'\
"$DATD"/Energy_data.csv | csplit -zf Energy_out_ -b "%03d.csv" -\
'%«««««START»»»»»»%' '/^«««««START»»»»»»/' {*}

for file in Energy_out_*
do
sed -i '/«««««FINISHED»»»»»»/q' $file
sed -i 's/«««««START»»»»»»//' $file
sed -i 's/«««««FINISHED»»»»»»//' $file
sed -i '/^$/d' $file

#Removing column labels, 1st row and last as they were having incomplete data

awk -F, ' $1 = ($1 >=50 ?  0 :  $1)  1' OFS=, $file > temp.csv
awk -F, ' $1 = ($1 <0 ?  0 :  $1)  1' OFS=, temp.csv > $file

#Computing total energy consumption for each application
```

```
awk -F, 'for (i=1;i<=NF;i++) sum[i]+=$i;; ENDfor (i in sum) printf "%d,",
sum[i];' $file »Energy_"$a"_"$b"_Final.csv

#Find number of rows in each file to compute the execution time of each
application (x Sampling_period)

wc -l $file | awk 'printf "%d", $1' »Energy_"$a"_"$b"_Final.csv
printf "\n" »Energy_"$a"_"$b"_Final.csv
done

mv Energy_"$a"_"$b"_Final.csv "$DIR_D"/
rm Energy_out_*
```

```
############################################################
#Boxplot.py
#Script for plotting the boxplots in Chapter 5
Created by Karunakar R. Basireddy (krb1g15@soton.ac.uk)
############################################################
import numpy
import numpy as np
import matplotlib as mpl
import xlrd

#Read the excel file
loc = ("/home/karunakar/<path>/file.xlsx")

#Font Size
mpl.rcParams.update({'font.size':  10})

# To open Workbook
wb = xlrd.open_workbook(loc)
sheet = wb.sheet_by_index(0)
xticks = numpy.arange(100, 1400, 100)
#print(sheet.ncols)
#print(sheet.nrows)
# For row 0 and column 0
#sheet.cell_value(0, 0)
data = [[sheet.cell_value(c, r)
for c in range(sheet.nrows)] for r in range(sheet.ncols)]
```

```
#dumping an array to a csv file
#numpy.savetxt("foo.csv", data, delimiter=",")
#print(data)

# agg backend is used to create plot as a .png file
mpl.use('agg')

import matplotlib.pyplot as plt

# Create a figure instance
fig = plt.figure(1, figsize=(7, 2.7))

# Create an axes instance
ax = fig.add_subplot(111)

#Adding grids
#ax.grid(True)
plt.gca().yaxis.grid(True)

# Create the boxplot
bp = ax.boxplot(data)
plt.title('big to LITTLE')
plt.xlabel('Number of Decision Stumps')
plt.ylabel('Percentage Error (%)')

# Custom x-axis labels
ax.set_xticklabels(xticks)

# Save the figure
fig.savefig('fig1.png', bbox_inches='tight')
```

References

[1] K. Rupp, "42 years of microprocessor trend data." https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, February 2018. [Online; accessed 16-Jan-2019].

[2] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, pp. 16–29, 2011.

[3] A. Das, G. Merrett, M. Tribastone, and B. Al-Hashimi, "Workload change point detection for run-time thermal management of embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–16, 2015.

[4] K. Choi, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for an MPEG player," *Journal of Low Power Electronics*, vol. 1, no. 1, pp. 27–43, 2005.

[5] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated power-performance optimization in manycores," in *Proceedings of the International conference on Parallel architectures and compilation techniques*, pp. 51–61, IEEE, 2013.

[6] ITRS Reports, "International Technology Roadmap for Semiconductors (ITRS)." http://www.itrs2.net/2011-itrs.html. [Online; accessed 26-Oct-2018].

[7] "Amdahl's law demonstration." https://en.wikipedia.org/wiki/Amdahl%27s_law. [Online; accessed 26-Oct-2018].

[8] Donald Firesmith, "Multicore processing." https://insights.sei.cmu.edu/sei_blog/2017/08/multicore-processing.html. [Online; accessed 26-Oct-2018].

[9] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proc. of the Linux Symposium*, vol. 2, pp. 215–230, sn, 2006.

[10] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ACM*

SIGARCH Computer Architecture News, vol. 40, pp. 213–224, IEEE Computer Society, 2012.

[11] "Odroid-XU3 Board Diagram." https://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127&tab_idx=2. [Online; accessed 31-Oct-2018].

[12] K. Crijns, "The Intel Xeon E5-2630V2 Review." https://uk.hardware.info/reviews/4790/3/intel-xeon-e5-2600-v2-review-ivy-bridge-ep-for-servers-50-more-cores. [Online; accessed 02-Nov-2018].

[13] "The Intel Xeon Phi MIC Architecture." http://miclab.pl/core/index.php/MICLAB:Architektura_Intel_MIC_-_Wprowadzenie. [Online; accessed 02-Nov-2018].

[14] S. Jarp, R. Jurga, and A. Nowak, "Perfmon2: a leap forward in performance monitoring," in *Journal of Physics: Conference Series*, vol. 119, p. 042017, IOP Publishing, 2008.

[15] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, vol. 2011, 2009.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, IEEE, 2009.

[17] H.-Q. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," 1999.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 24–36, ACM, 1995.

[19] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "SPARTA: runtime task allocation for energy efficient heterogeneous many-cores," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, p. 27, ACM, 2016.

[20] R. A. Shafik, S. Yang, A. Das, L. A. Maeda-Nunez, G. V. Merrett, and B. M. Al-Hashimi, "Learning transfer-based adaptive energy minimization in embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 877–890, 2016.

[21] A. S. Tanenbaum, *Structured computer organization*. Prentice Hall PTR, 1984.

[22] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[23] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[24] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[25] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012.

[26] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo: Making the case for an ARM-based HPC system," *Future Generation Computer Systems*, vol. 36, pp. 322–334, 2014.

[27] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC?," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 40, ACM, 2013.

[28] "Odroid-XU3." www.hardkernel.com/main/products.

[29] "Juno r2." https://www.arm.com/files/pdf/Juno_r2_datasheet.pdf.

[30] "Mediatek X20." http://www.96boards.org/product/mediatek-x20/.

[31] R. A. Shafik, B. M. Al-Hashimi, and J. S. Reeve, "System-level design optimization of reliable and low power multiprocessor system-on-chip," *Microelectronics Reliability*, vol. 52, no. 8, pp. 1735–1748, 2012.

[32] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, pp. 746–749, ACM, 2007.

[33] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, *et al.*, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, 2008.

[34] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, C. Watnik, A. T. Tran, Z. Xiao, *et al.*, "A 167-processor computational platform in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1130–1144, 2009.

[35] A. Das, B. M. Al-Hashimi, and G. V. Merrett, "Adaptive and hierarchical runtime manager for energy-aware thermal management of embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 2, p. 24, 2016.

[36] D. Flynn, "An ARM perspective on addressing low-power energy-efficient soc designs," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 73–78, ACM, 2012.

[37] Y. Tan, W. Liu, and Q. Qiu, "Adaptive power management using reinforcement learning," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, pp. 461–467, ACM, 2009.

[38] J. Pouwelse, K. Langendoen, and H. J. Sips, "Application-directed voltage scaling," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, no. 5, pp. 812–826, 2003.

[39] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 299–316, 2000.

[40] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive DVFS and thread packing under power caps," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 175–185, 2011.

[41] C.-H. Hsu and U. Kremer, "Compiler-directed dynamic voltage scaling for memory-bound applications," *Technical Report DCS-TR-498, Department of Computer Science, Rutgers University*, 2002.

[42] J. Luo and N. K. Jha, "Power-efficient scheduling for heterogeneous distributed real-time embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 6, pp. 1161–1170, 2007.

[43] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *Proceedings of the 45th annual design automation conference*, pp. 191–196, ACM, 2008.

[44] M. Qiu and E. H.-M. Sha, "Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 2, p. 25, 2009.

[45] L. K. Goh, B. Veeravalli, and S. Viswanathan, "Design of fast and efficient energy-aware gradient-based scheduling algorithms heterogeneous embedded multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 1, pp. 1–12, 2009.

[46] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *SIGARCH Computer Architecture News*, vol. 39, pp. 449–460, ACM, 2011.

[47] A. Weissel and F. Bellosa, "Process cruise control: event-driven clock scaling for dynamic power management," in *Proc. of international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 238–246, ACM, 2002.

[48] L. C. Singleton, C. Poellabauer, and K. Schwan, "Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling," in *Electronic Imaging*, pp. 121–125, International Society for Optics and Photonics, 2005.

[49] V. Spiliopoulos, G. Keramidas, S. Kaxiras, and K. Efstathiou, "Power-performance adaptation in intel core i7," 2011.

[50] A. Aalsaud, R. Shafik, A. Rafiev, F. Xia, S. Yang, and A. Yakovlev, "Power-aware performance adaptation of concurrent applications in heterogeneous many-core systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 368–373, ACM, 2016.

[51] J. Ma, G. Yan, Y. Han, and X. Li, "An analytical framework for estimating scale-out and scale-up power efficiency of heterogeneous manycores," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 367–381, 2016.

[52] E. Del Sozzo, G. Durelli, E. Trainiti, A. Miele, M. Santambrogio, and C. Bolchini, "Workload-aware power optimization strategy for asymmetric multiprocessors," in *2016 Design, Automation & Test in Europe Conference & Exhibition*, pp. 531–534, IEEE, 2016.

[53] B. K. Reddy, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Online concurrent workload classification for multi-core energy management," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 621–624, March 2018.

[54] K. R. Basireddy, E. Weber Wachter, B. Al-Hashimi, and G. Merrett, "Memory and thread synchronization contention-aware dvfs for hpc systems," in *Adaptive Many-Core Architectures and Systems Workshop*, 2018.

[55] K. R. Basireddy, E. W. Wachter, B. M. Al-Hashimi, and G. V. Merrett, "Workload-aware runtime energy management for hpc systems," in *The 2018 International Conference on High Performance Computing & Simulation (HPCS)*, p. 8, 2018.

[56] B. K. Reddy, A. K. Singh, G. V. Merrett, and B. M. Al-Hashimi, "ITMD: Run-time management of concurrent multi-threaded applications on heterogeneous multi-cores," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[57] B. K. Reddy, A. Singh, D. Biswas, G. Merrett, and B. Al-Hashimi, "Inter-cluster thread-to-core mapping and dvfs on heterogeneous multi-cores," *IEEE Transactions on Multiscale Computing Systems*, pp. 1–14, 2017.

[58] K. R. Basireddy, A. K. Singh, G. Merrett, and B. Al-Hashimi, "AdaMD: Adaptive Mapping and DVFS on Heterogeneous multi-cores," ***submitted to*** *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 11, 2019.

[59] A. K. Singh, K. R. Basireddy, P. Alok, G. Merrett, and B. Al-Hashimi, "Energy-efficient collaborative adaptation in heterogeneous mobile SoCs," ***submitted to*** *IEEE Transactions on Computers*, p. 11, 2018.

[60] E. Weber Wachter, C. d. Bellefroid, K. R. Basireddy, A. K. Singh, B. Al-Hashimi, and G. Merrett, "Predictive thermal management for energy-efficient execution of concurrent applications on heterogeneous multi-cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, p. 12, 2019.

[61] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent opencl applications on cpu-gpu mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 147, 2017.

[62] A. K. Singh, C. Leech, B. K. Reddy, B. M. Al-Hashimi, and G. V. Merrett, "Learning-based run-time power and energy management of multi/many-core systems: current and future trends," *Journal of Low Power Electronics*, vol. 13, no. 3, pp. 310–325, 2017.

[63] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux HPC Revolution*, vol. 45, 2002.

[64] Rob Oshana, "The keys to success in multicore application development." https://www.embedded.com/design/operating-systems/4008333/2/The-keys-to-success-in-multicore-application-development, August 2009. [Online; accessed 25-Oct-2018].

[65] M. Wolf, *Computers as components: principles of embedded computing system design*. Elsevier, 2012.

[66] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, no. 3, p. 034008, 2008.

[67] R. Cochran, C. Hankendi, A. Coskun, and S. Reda, "Identifying the optimal energy-efficient operating points of parallel workloads," in *Computer-Aided Design, 2011 IEEE/ACM International Conference on*, pp. 608–615, 2011.

[68] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*, pp. 1–8, 2010.

[69] T. D. Burd and R. W. Brodersen, "Energy efficient CMOS microprocessor design," in *Proc. of the Twenty-Eighth Hawaii International Conference on System Sciences*, vol. 1, pp. 288–297, IEEE, 1995.

[70] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Mobile Computing*, pp. 449–471, Springer, 1994.

[71] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: A platform for os-level power management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, pp. 289–302, 2009.

[72] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of NP-completeness," *WH Free. Co., San Fr*, pp. 90–91, 1979.

[73] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, 2009.

[74] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, "An industrial design space exploration framework for supporting run-time resource management on multi-core systems," in *Design, Automation & Test in Europe Conference & Exhibition, 2010*, pp. 196–201, IEEE, 2010.

[75] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[76] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–10, ACM, 2013.

[77] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[78] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, 2008.

[79] A. K. Singh, "Run-time mapping techniques for noc-based heterogeneous mpsoc platforms," *PHD. Nanyang Technological University*, 2013.

[80] "Multi-core chips by academia and industry." https://en.wikipedia.org/wiki/Multi-core_processor. [Online; accessed 26-Oct-2018].

[81] A. Agarwal, "Raw computation," *Scientific American*, vol. 281, no. 2, pp. 60–63, 1999.

[82] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. Lai, J. W. Webb, E. W. Work, D. Truong, T. Mohsenin, and B. M. Baas, "Asap: An asynchronous array of simple processors," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 3, pp. 695–705, 2008.

[83] S. W. Keckler, D. Berger, M. Dahlin, L. John, C. Lin, K. McKinley, T. Keller, and K. Nowka, "Tera-op reliable intelligently adaptive processing system (trips)," tech. rep., TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES, 2004.

[84] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert, "Hs-scale: a hardware-software scalable mp-soc architecture for embedded systems," in *VLSI, 2007. ISVLSI'07. IEEE Computer Society Annual Symposium on*, pp. 21–28, IEEE, 2007.

[85] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 291, IEEE Computer Society, 2003.

[86] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, *et al.*, "Tile64-processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 88–598, IEEE, 2008.

[87] S. Richardson, "Mpoc: A chip multiprocessor for embedded systems," *HP Laboratories Technical Report HPL-2002*, vol. 186, 2002.

[88] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 24–33, 2008.

[89] G. J. Smit, A. B. Kokkeler, P. T. Wolkotte, and M. D. Van De Burgwal, "Multi-core architectures and streaming applications," in *Proceedings of the 2008 international workshop on System level interconnect prediction*, pp. 35–42, ACM, 2008.

[90] "4s-smart chips for smart surroundings." https://www.utwente.nl/en/eemcs/caes/. [Online; accessed 26-Oct-2018].

[91] T. Arpinen, P. Kukkala, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, "Configurable multiprocessor platform with RTOS for distributed execution of UML 2.0 designed applications," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pp. 1324–1329, European Design and Automation Association, 2006.

[92] H. Nikolov, T. Stefanov, and E. Deprettere, "Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, p. 726096, 2008.

[93] "Cortex-a15 technical reference manual." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/DDI0438I_cortex_a15_r4p0_trm.pdf.

[94] "Cortex-a7 technical reference manual." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/DDI0464F_cortex_a7_mpcore_r0p5_trm.pdf.

[95] D. Brodowski, "Current trend in linux kernel power management, linuxtag 2005," 2005.

[96] W. Yuan and K. Nahrstedt, "Practical voltage scaling for mobile multimedia devices," in *Proceedings of the 12th Annual International Conference on Multimedia*, pp. 924–931, ACM, 2004.

[97] Y. Gu and S. Chakraborty, "Control theory-based DVS for interactive 3D games," in *Proceedings of the 45th annual Design Automation Conference*, pp. 740–745, ACM, 2008.

[98] S. Wildermann, T. Ziermann, and J. Teich, "Game-theoretic analysis of decentralized core allocation schemes on many-core systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1498–1503, EDA Consortium, 2013.

[99] R. Jejurikar and R. Gupta, "Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems," in *Proceedings of the 2004 international symposium on Low power electronics and design*, pp. 78–81, ACM, 2004.

[100] H. Jung and M. Pedram, "Continuous frequency adjustment technique based on dynamic workload prediction," in *VLSI Design, 21st International Conference on*, pp. 249–254, IEEE, 2008.

[101] M. Pedram, "Power optimization and management in embedded systems," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pp. 239–244, ACM, 2001.

[102] J. Liu and P. H. Chou, "Optimizing mode transition sequences in idle intervals for component-level and system-level energy minimization," in *Proceedings of the 2004 International Conference on Computer-Aided Design*, pp. 21–28, ACM, 2004.

[103] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proceedings of the 2004 international symposium on Low power electronics and design*, pp. 174–179, ACM, 2004.

[104] S. Yue, D. Zhu, Y. Wang, and M. Pedram, "Reinforcement learning based dynamic power management with a hybrid power supply," in *Computer Design, 30th International Conference on*, pp. 81–86, IEEE, 2012.

[105] H. Shen, Y. Tan, J. Lu, Q. Wu, and Q. Qiu, "Achieving autonomous power management using reinforcement learning," *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 2, p. 24, 2013.

[106] L. A. Maeda-Nunez, A. K. Das, R. A. Shafik, G. V. Merrett, and B. Al-Hashimi, "Pogo: an application-specific adaptive energy minimisation approach for embedded systems," 2015.

[107] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 692–707, 2010.

[108] W. Quan and A. D. Pimentel, "A scenario-based run-time task mapping algorithm for mpsocs," in *Proceedings of the 50th Annual Design Automation Conference*, p. 131, ACM, 2013.

[109] R. Ge, X. Feng, W.-c. Feng, and K. W. Cameron, "Cpu miser: A performance-directed, run-time system for power-aware clusters," in *Parallel Processing, 2007. ICPP 2007. International Conference on*, pp. 18–18, IEEE, 2007.

[110] C.-h. Hsu and W.-c. Feng, "A power-aware run-time system for high-performance computing," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 1, IEEE Computer Society, 2005.

[111] X. Feng, R. Ge, and K. W. Cameron, "Power and energy profiling of scientific applications on distributed systems," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 10–pp, IEEE, 2005.

[112] B. Rountree, D. K. Lownenthal, B. R. De Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: making DVS practical for complex HPC applications," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 460–469, ACM, 2009.

[113] G. Lawson, V. Sundriyal, M. Sosonkina, and Y. Shen, "Runtime power limiting of parallel applications on intel xeon phi processors," in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, pp. 39–45, 2016.

[114] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs," in *SC 2006 conference, proceedings of the ACM/IEEE*, pp. 14–14, IEEE, 2006.

[115] V. W. Freeh and D. K. Lowenthal, "Using multiple energy gears in MPI programs on a power-scalable cluster," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 164–173, ACM, 2005.

[116] M. Etinski, J. Corbalan, J. Labarta, M. Valero, and A. Veidenbaum, "Power-aware load balancing of large scale MPI applications," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[117] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained HPC applications," in *International conference on high performance computing*, pp. 394–408, Springer, 2015.

[118] Z. Lai, K. T. Lam, C.-L. Wang, and J. Su, "Latency-aware dvfs for efficient power state transitions on many-core architectures," *J. Supercomput.*, vol. 71, no. 7, pp. 2720–2747, 2015.

[119] M. S. Vaibhav Sundriyal, "Runtime power-aware energy-saving scheme for parallel applications," in *Iowa State University Computer Science Technical Reports*, p. 17, 2015.

[120] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[121] P. G. Emma, "Understanding some simple processor-performance limits," *IBM journal of Research and Development*, vol. 41, no. 3, pp. 215–232, 1997.

[122] W. Quan and A. D. Pimentel, "A hybrid task mapping algorithm for heterogeneous mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 1, p. 14, 2015.

[123] S. Yang, R. A. Shafik, G. V. Merrett, E. Stott, J. M. Levine, J. Davis, and B. M. Al-Hashimi, "Adaptive energy minimization of embedded heterogeneous systems using regression-based learning," in *Power and Timing Modeling, Optimization and Simulation, 2015 25th International Workshop on*, pp. 103–110, IEEE, 2015.

[124] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, "Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 123, 2017.

[125] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd conference on Computing frontiers*, pp. 29–40, ACM, 2006.

[126] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 927–930, ACM, 2009.

[127] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, pp. 125–138, ACM, 2010.

[128] "Exynos 5 octa (5422)." www.samsung.com/exynos/.

[129] "ARM big.LITTLE Technology." http://www.arm.com/.

[130] "System Administration for the Intel Xeon Phi Coprocessor." http://software.intel.com/sites/default/files/article/373934/system-administration-for-the-intel-xeon-phi-coprocessor.pdf. [Online; accessed 02-Nov-2018].

[131] "Intel Manycore Platform Software stack (MPSS)." http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss. [Online; accessed 02-Nov-2018].

[132] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001.

[133] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *Workload Characterization Symposium, 2005. Proceedings of the International*, pp. 34–45, IEEE, 2005.

[134] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 412–423, 2007.

[135] R. Longbottom, "Roy Longbottom's PC Benchmark Collection." http://www.roylongbottom.org.uk, September 2014. [Online; accessed 2-June-2015].

[136] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proc. of the 1996 Annu. Conf. on USENIX Annual Technical Conference*, ATEC '96, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 1996.

[137] F. Eibe, M. Hall, and I. Witten, "The weka workbench. online appendix for" data mining: Practical machine learning tools and techniques," *Morgan Kaufmann*, 2016.

[138] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[139] W. Yuan, K. Nahrstedt, S. Adve, D. L. Jones, and R. H. Kravets, "Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems," in *Multimedia Computing and Networking 2003*, vol. 5019, pp. 1–14, International Society for Optics and Photonics, 2003.

[140] "Odroid-XU3." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0273a/ar01s04.html.

[141] M. J. Walker, S. Diestelhorst, A. Hansson, A. K. Das, S. Yang, B. M. Al-Hashimi, and G. V. Merrett, "Accurate and stable run-time power modeling for mobile and embedded cpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 106–119, 2017.

[142] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[143] S.-H. Kim and B. L. Nelson, "Selecting the best system," *Handbooks in operations research and management science*, vol. 13, pp. 501–534, 2006.

[144] K. Najarian and R. Splinter, *Biomedical signal and image processing*. CRC press, 2005.

[145] T. W. Liao, "Clustering of time series data-a survey," *Pattern recognition*, vol. 38, no. 11, pp. 1857–1874, 2005.

[146] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, pp. 30–31 and 280–288. Elsevier, 4 ed., 2008.

[147] M. A. T. Figueiredo and A. K. Jain, "Unsupervised learning of finite mixture models," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 3, pp. 381–396, 2002.

[148] J. Mao and A. K. Jain, "A self-organizing network for hyperellipsoidal clustering (hec)," *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 16–29, 1996.

[149] S. Sinha, J. Suh, B. Bakkaloglu, and Y. Cao, "Workload-aware neuromorphic design of the power controller," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 3, pp. 381–390, 2011.

[150] A. S. Bischoff, *User-experience-aware system optimisation for mobile systems*. PhD thesis, University of Southampton, 2016.

[151] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," in *Proceedings of the 7th International Conference on Computing Frontiers*, pp. 287–296, ACM, 2010.

[152] A. Das, S. Ozdemir, G. Memik, and A. Choudhary, "Evaluating voltage islands in cmps under process variations," in *Computer Design, 25th International Conference on*, pp. 129–136, IEEE, 2007.

[153] F. Kong, W. Yi, and Q. Deng, "Energy-efficient scheduling of real-time tasks on cluster-based multicores," in *DATE*, pp. 1–6, IEEE, 2011.

[154] "Memcopy." https://github.com/ARMsoftware/workload-automation/blob/a747ec7e4c2ea8a25bfc675f80042eb6600c7050/wlauto/workloads/memcpy/src/jni/memcopy.c.

[155] "XDA-developersforums." https://forum.xda-developers.com/general/general/ref-to-date-guide-cpu-governors-o-t3048957.

[156] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan, "Survey of techniques and architectures for designing energy-efficient data centers," *IEEE Systems Journal*, vol. 10, no. 2, pp. 507–519, 2016.

[157] J. Choi, M. Dukhan, X. Liu, and R. Vuduc, "Algorithmic time, energy, and power on candidate hpc compute building blocks," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 447–457, IEEE, 2014.

[158] B. Li, H.-C. Chang, S. Song, C.-Y. Su, T. Meyer, J. Mooring, and K. W. Cameron, "The power-performance tradeoffs of the intel xeon phi on HPC applications," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 1448–1456, IEEE, 2014.

[159] J. Wood, Z. Zong, Q. Gu, and R. Ge, "Energy and power characterization of parallel programs running on intel xeon phi," in *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pp. 265–272, IEEE, 2014.

[160] N. Manchanda and K. Anand, "Non-uniform memory access (numa)," *New York University*, vol. 4, 2010.

[161] G. Liu, J. Park, and D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *Computer Design, 31st International Conference on*, pp. 54–61, IEEE, 2013.

[162] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pp. 23–32, IEEE, 2009.

[163] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, "Evaluation of cpu frequency transition latency," *Computer Science-Research and Development*, vol. 29, no. 3-4, pp. 187–195, 2014.

[164] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, and S. Poole, "Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system," *Sustainable Computing: Informatics and Systems*, vol. 5, pp. 14–30, 2015.

[165] P. Greenhalgh, "big.LITTLE processing with ARM cortex-a15 & cortex-a7," *ARM White paper*, pp. 1–8, 2011.

[166] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative CPU-GPU thermal management," in *Proceedings of the Annual Design Automation Conference*, p. 47, ACM, 2016.

[167] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Proc. of Ottawa Linux Symposium*, pp. 269–288, Citeseer, 2006.

[168] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[169] K. Yu, D. Han, C. Youn, S. Hwang, and J. Lee, "Power-aware task scheduling for big. LITTLE mobile processor," in *SoC Design Conference, 2013 International*, pp. 208–212, IEEE, 2013.

[170] "Linux-governors." https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[171] D. Stamoulis and D. Marculescu, "Can we guarantee performance requirements under workload and process variations?," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 308–313, ACM, 2016.

[172] P. K. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc)," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 212–217, ACM, 2008.

[173] E. W. Brião, D. Barcelos, and F. R. Wagner, "Dynamic task allocation strategies in mpsoc for soft real-time applications," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1386–1389, ACM, 2008.

[174] J. Huang, A. Raabe, C. Buckl, and A. Knoll, "A workflow for runtime adaptive task allocation on heterogeneous mpsocs," in *Design, Automation and Test in Europe (DATE)*, 2011.

[175] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pp. 1–10, IEEE, 2013.

[176] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto, "Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 2, p. 6, 2012.

[177] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[178] N. Duffy and D. Helmbold, "Boosting methods for regression," *Machine Learning*, vol. 47, no. 2-3, pp. 153–200, 2002.

[179] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[180] B. K. Reddy, M. J. Walker, D. Balsamo, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Empirical CPU power modelling and estimation in the gem5 simulator," in *IEEE Intl. Symp. on Power and Timing Modeling, Optimization and Simulation*, pp. 1–8, 2017.

[181] P. D. Bryan, J. G. Beu, T. M. Conte, P. Faraboschi, and D. Ortega, "Our many-core benchmarks do not use that many cores," *System*, vol. 6, p. 8, 2009.