Power, Performance, and Energy Management of Heterogeneous Architectures

by

Chetan Arvind Patil

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved August 2019 by the
Graduate Supervisory Committee:

Umit Y. Ogras, Chair
Chaitali Chakrabarti
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

December 2019

ABSTRACT

Many core modern multiprocessor systems-on-chip offers tremendous power and performance optimization opportunities by tuning thousands of potential voltage, frequency and core configurations. Applications running on these architectures are becoming increasingly complex. As the basic building blocks, which make up the application, change during runtime, different configurations may become optimal with respect to power, performance or other metrics. Identifying the optimal configuration at runtime is a daunting task due to a large number of workloads and configurations. Therefore, there is a strong need to evaluate the metrics of interest as a function of the supported configurations.

This thesis focuses on two different types of modern multiprocessor systems-on-chip (SoC): Mobile heterogeneous systems and tile based Intel Xeon Phi architecture.

For mobile heterogeneous systems, this thesis presents a novel methodology that can accurately instrument different types of applications with specific performance monitoring calls. These calls provide a rich set of performance statistics at a basic block level while the application runs on the target platform. The target architecture used for this work (Odroid XU3) is capable of running at 4940 different frequency and core combinations. With the help of instrumented application vast amount of characterization data is collected that provides details about performance, power and CPU state at every instrumented basic block across 19 different types of applications. The vast amount of data collected has enabled two runtime schemes. The first work provides a methodology to find optimal configurations in heterogeneous architecture using classifiers and demonstrates an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively. The second work using same data shows a novel imitation learning framework for dynamically controlling the type, number, and the frequencies of active cores to achieve an average of 109% PPW improvement compared to the default governors.

This work also presents how to accurately profile tile based Intel Xeon Phi architecture while training different types of neural networks using open image dataset on deep learning framework. The data collected allows deep exploratory analysis. It also showcases how different hardware parameters affect performance of Xeon Phi.

Dedicated to my family

ACKNOWLEDGEMENTS

This thesis is the result of efforts and guidance from many people who in all capacity contributed to the research work in this thesis.

Grateful to Dr. Umit Y. Ogras, my research advisor for his patience, advice, and guidance over the years which led to this thesis work. Without his support, this thesis would not be possible. Thank you to Dr. Chaitali Chakrabarti and Dr. Aviral Shrivastava for being on my thesis committee.

I also would like to thank Ganapati Bhat, Sumit K. Mandal, and Ujjwal Gupta for their support and help with Odroid. Their guidance around Odroid XU3 helped in developing elegant solutions leading to work getting published in journals. I greatly enjoyed working with Dr. Partha Pratim Pande and Dr. Janardhan Rao Doppa. Their inputs on imitation learning helped in getting deep insight into power management solution. I also cherish the research work done on multi-core architectures with Dr. Prabhat Mishra and Subodha Charles.

Thankful to Arizona State University for providing an environment that made the process both productive and enjoyable at the same time. My years at ASU also allowed me to interact with numerous students which have helped me grow professionally and made my journey at ASU memorable.

Finally, thank you to my parents Arvind Pitambar Patil and Kalpana Arvind Patil for their endless support and understanding.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

State-of-the-art smartphones and tablets have to satisfy the performance requirements of a diverse range of applications under tight power and thermal budget [15, 21, 64]. The number of power management configurations offered by modern multiprocessor system-on-chip (MpSoCs), such as the number of voltage-frequency levels and active cores, have been increasing steadily to adapt to these dynamically varying requirements [52]. For example, octa-core big.LITTLE architectures have 20 different CPU core configurations that can be selected at runtime. Combined with the voltage and frequency levels, this leads to more than 4000 dynamic configurations to consider during optimization.

The large design space results in more than one order of magnitude variation in both

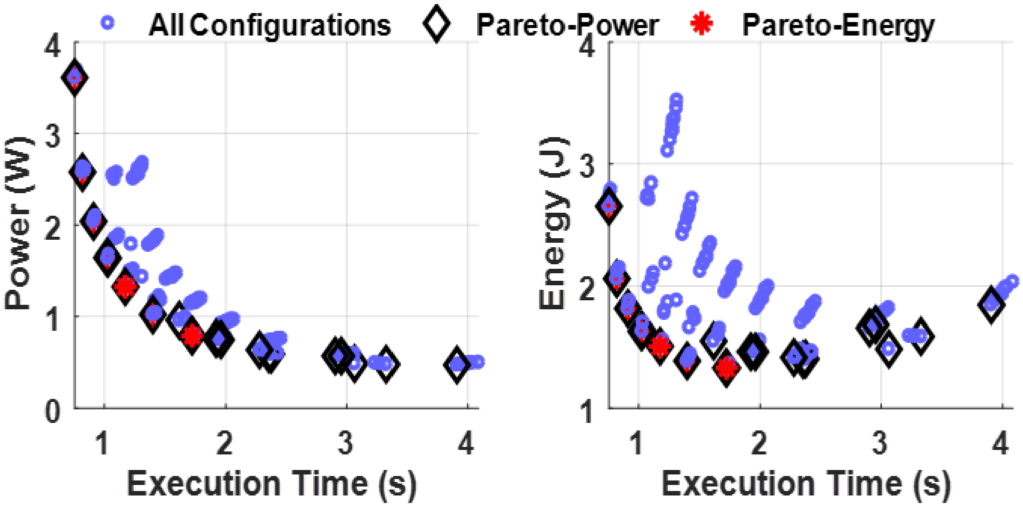


Figure 1.1: 256 different frequency and core configurations of the Blackscholes application showing the trade-off between (a) power consumption and execution time, (b) energy consumption and execution time [22].

power consumption and performance, as shown in Figure 1.1(a). Moreover, the definition of the optimality can change depending on the context. For instance, users prefer to maximize the responsiveness (i.e., performance) for interactive applications, while minimizing the energy becomes the priority when the platform is running out of power. Therefore, it is crucial to identify the optimal configuration at runtime.

Chip designers and power management architects spend significant effort to attain the optimal power-performance trade-off. For example, Figure 1.1(a) plots power consumption and execution time of a multi-threaded application for 128 different core and operating frequency configurations. It can be seen that many configurations are close to the Pareto-optimal curve. Frequency governors integrated in the OS-stack leverage this fact effectively to deliver the desired trade-off [2, 36]. For instance, the interactive and on-demand governors increase the frequency whenever core utilizations exceed a threshold to maximize the performance, while the powersave governor chooses the minimum operating frequency to minimize power consumption [54]. Similarly, the dynamic power management algorithms, such as cpuidle, increase (decrease) the number of active cores when the core utilizations are above (below) tunable thresholds [4, 53]. Hence, these highly optimized governors can dynamically scale the number of active cores and frequency to optimize the power-performance trade-off. However, none of these approaches can guarantee optimality with respect to other metrics, such as energy consumption. For instance, Figure 1.1(b) shows that many Pareto-optimal configurations in the power-performance plane are far away from the *Pareto curve in the energy-performance plane*. Moreover, a governor that chooses the lowest power configuration results in 39% more energy consumption and 126% slower execution with respect to the minimum energy configuration.

Despite the impressive progress in hardware support, the majority of existing solutions in commercial mobile platforms still rely on simple heuristic algorithms whose simplicity is hard to beat. For example, interactive and ondemand governors in mobile systems increase (decrease) the operating frequency by one level when the utilization exceeds (goes below) a static threshold. They are very aggressive in driving the frequency levels up since one of the main differentiating factors is still the performance perceived by the user, such as touch responsiveness or the maximum throughput while running key performance indicators (KPI). Hence, this choice leads to running the applications at the highest frequencies during most of their active time, leaving little room for improving the performance. In contrast, the powersave governor uses the lowest frequency levels to minimize power consumption. However, this choice does not necessarily maximize the energy-efficiency [22]. Therefore, new solutions must *not only maximize the energy-efficiency*, but also need to have negligible runtime overhead to be *practical*.

To outperform default governors that are de-facto across all mobile platforms, it is crucial to have insightful data from numerous applications which not only considers different core and frequency configuration of target platform but also presents deep insight into different phases of the application and data around it [8]. To enable algorithms that can choose the optimal configuration with respect to a given metric as a function of the workload, the instrumentation framework and data collection are vital. To this end, **Chapter 3** presents a novel approach for instrumenting different types of applications precisely by considering different phases they are made of. It also provides details on how the instrumented framework can be easily integrated with any to be proposed runtime methodology. This approach is illustrated using 19 commonly used benchmarks.

Chapter 4 presents two novel and practical dynamic resource management technique for heterogeneous mobile platforms that made use of the proposed instrumentation framework and data collected to first enable runtime optimal configuration selection and then using the same data to outperform the first strategy by making use of the imitation learning (IL) framework [39, 59, 61, 66]. The proposed technique controls the type (Big/Little), number, and the frequencies of active cores. IL is a promising approach since it enables us to automatically transform an optimal offline solution into an efficient online policy. The phase-level instrumentation has enabled the construction of a near-optimal Oracle policy *offline* by exploiting characterization data for target applications available at design time. Both the proposed solution employ advanced algorithms to design low-overhead control policies that outperform the default ondemand, interactive and performance governors.

Apart from mobile based MpSoCs, many core architectures that are designed to assist deep learning have also attracted significant attention. The goal of these architectures is to provide performance to enable faster completion of the task rather than focusing on efficiency. **Chapter 5** explores the Xeon Phi [65] architecture equipped with 64 cores and capable of running different types of clustering, memory, and thread modes. This study provides a deep insight into the tile based heterogeneous Xeon Phi architecture by profiling widely used neural networks [40] [28] [63] [67] and also showcase how to profile networks during runtime to collect data using performance counters and power consumption sensor. Chapter 5 presents the results of this exploratory study concerning runtime, cycles per instruction, power consumption and memory bandwidth.

Chapter 2

RELATED RESEARCH

Advances in heterogeneous processors have enabled widespread adoption of mobile platforms. These platforms integrate multiple types of cores (Big/Little), graphics processing units, and other accelerators to support a wide variety of applications [21, 25, 41]. The heterogeneity in mobile platforms requires new techniques for their resource management at runtime [12, 20, 30, 35, 50, 56, 72]. To develop accurate runtime optimization scheme, phase-level details of the application on target architecture is necessary. Phase-level performance and power analysis provide fine-grained and reliable information about the workload. This information enables accurate power and performance models across different platforms [78] and practical power management algorithms [32]. For example, using the phase-level analysis one can collect statistics on one platform and use it to predict the power and performance on another platform [78]. This leads to significant improvements in the accuracy of the models by using this insight compared to an approach that uses aggregate application statistics. Therefore, in contrast to the other phase-level approaches, this work leverages the use of phase-level for characterization for many benchmarks which is then used to develop runtime schemes [22, 24, 45] that outperform state-of-the art governors. **Section 3.3** provides more details on how phase-level instrumentation is implemented and used for building classifiers and Oracle based policy that perform much better than the recently proposed algorithm [1].

Default governors on mobile platforms, such as interactive and ondemand governors [54], and algorithms [3, 50, 56] take management decisions based on the system utilization. However, decisions made with utilization alone may not be optimal in terms of PPW or energy consumption. Therefore, recent research has focused on new algorithms for runtime management of mobile platforms [1, 5, 17, 18, 22, 58]. However, studies presented in [1, 5, 18, 58] do not use phase level instrumentation, which provides fine-grained information to take optimal power management decisions. The technique in [18] chooses only the type of core (Big/Little) on which the task has to be executed. In the proposed power management technique [22, 45], by controlling four configuration knobs, namely, the number of active big cores, number of active little cores and their corresponding frequencies, is used to better existing runtime techniques on heterogeneous architectures. Controlling more knobs allow us to explore number of decisions. Furthermore, the work presented in [1] uses application-specific policies, which is not scalable for unseen applications. A recent study uses a gradient search approach to decide the operating configuration of the platform such that temperature violations are minimized [5, 74]. Specifically, it reduces the frequencies iteratively until the temperature constraint is met. Therefore, it takes a few iterations of decision making for the approach to achieve its objective. **Chapter 4** showcases different approaches [22, 45] that take decision at each snippet using the classifier and also using learned machine learning models. The technique proposed in uses phase level instrumentation.

Widespread use of mobile platforms in the last decade is enabled by advanced power management techniques, including dynamic core and uncore scaling [12, 38, 51], cache reconfiguration, task partitioning, task scheduling, and power budgeting [21, 29, 72, 75]. Significant number of these power management techniques focus on power and performance optimization through dynamic power management (DPM) and dynamic voltage, frequency scaling (DVFS). DPM consists of a set of algorithms that selectively turns off system components that are idle, such as controlling the number of active cores in the system depending on their utilization [4]. Similarly, DVFS-based schemes control the operating frequency of a core based on the utilization [30, 50, 54]. For example, millions of commercial mobile platforms run the ondemand and interactive governors [54]. However, these techniques do not guarantee optimality with respect to a given metric such as energy consumption. These approaches typically perturb the configuration by a single predetermined step. For example, interactive and ondemand governors increase (or decrease) the frequency of the processor if the utilization is above (below) a certain threshold [54]. The work presented in [76] proposes a technique to maximize the performance within a given power budget by estimating Pareto-optimal solutions dynamically. This approach relies on analytical power consumption and instructions per second model to find the Pareto-optimal frequency configurations of homogenous architectures. Approach presented in **Section 4.2** finds the Pareto-optimal core and frequency configuration in heterogeneous architectures using an extensive set of hardware measurements and multinomial logistic regression. Hence, this approach combines DVFS and DPM by setting the operating frequency/voltage and the type and number of active cores simultaneously.

In recent work, machine learning techniques are employed for dynamic resource management in mobile platforms [47, 55]. For example, the work presented in [55] builds offline decision trees for choosing the frequency of CPU and GPU. These models are then used at runtime to assign CPU and GPU frequencies for gaming applications. However, they do not consider the problem of choosing the number of active CPU cores. More recently, RL algorithms based on Q-table approach have been proposed for dynamic voltage and frequency selection [13, 71]. However, prior RL based solutions have several drawbacks. First, they are not practical as the size of the Q-table grows exponentially with the state and action space, as demonstrated in experimental results section. Thus, they require huge storage, which is limited to mobile platforms. Second, the performance of RL methods depend critically on the design of a good reward function to drive the learning process, which is a non-trivial task [19, 24]. Finally, the overall learning process to create near-optimal policies is very slow due to the need to try different actions at each state (i.e., exploration) to discover the best decision. An IL-based power management technique has been proposed for homogeneous manycore chips [39]. However, the proposed method is *not* applicable to heterogeneous mobile processors with different types of cores. Moreover, it is evaluated in a simulation setup for a specific suite of applications. Using an extensive experimental evaluation on a heterogeneous mobile hardware platform **Section 4.3** work [45] solves the problem by covering all possible core configuration for offline characterization and extends IL-based power management to heterogeneous architectures.

Many core architecture also play crucial role in deep learning research. Deep learning frameworks like Caffe [33] make it easy to train networks [40, 73] using open data set ImageNet ILSVRC [60]. Intel Xeon Phi architecture [65] provides many features that allow faster execution of application. In [11] authors explored different memory and cluster modes on Xeon Phi with respect to network-on-chip traffic. Authors in [68] provide comparative analysis of Xeon Phi against GPU and CPU for domain specific usage. Results [68] show that the performance on a many core integrated architecture in some cases is comparable or better than that on a GPU. When it comes to accessing data irregularly, GPU does outperform Xeon Phi and other CPU based architecture. **Chapter 5** provides detailed exploratory analysis of Xeon Phi architecture.

Chapter 3

WORKLOAD INSTRUMENTATION FRAMEWORK FOR HETEROGENEOUS MPSOCS

3.1 Overview

This chapter presents comprehensive methodology on how workload can be instrumented to obtain phase level power and performance characterization data on modern MpSoCs with negligible overhead using compiler infrastructure, API calls and power-performance counters on target architecture. The phase level instrumentation and characterization proposed in this chapter is not only unique, but easy to use across application. It is implemented and tested using widely referenced benchmarking suite covering different domains with mix of single, multi-threaded and graphic benchmarks. This chapter also showcases how the proposed instrumentation technique and data generated help in developing state-of-the-art dynamic power saving solutions on MpSoCs.

Major contributions:

Phase-Level Application Instrumentation (Section 3.3): Usage of applications on mobile architectures is growing exponentially and leading to more complex application. These applications are built of different phases which demand different requirements from system during the time they are active. The growing number of cores in MpSoCs also provides the ability to help these applications by enabling architecture resources that can adapt to the requirements of the application phase being executed. Widely used approaches to obtain power and performance counter based data from any architecture rely on running application as whole and then profiling it during its life on the target architecture. Such techniques provide time based profiling and not phase level profiling. The work presented in this section solves this problem by proposing, implementing and testing a framework developed using LLVM [43] compiler and PAPI calls [49]. Overhead added with this approach is less than 1% and also allows data collection using a different frequency and core knobs of MpSoCs. This is the first step in phase level workload instrumentation and characterization.

Data Characterization Methodology (Section 3.5): The second step after accurately instrumenting wide range of workloads is to collect data on the target platform. The collected data provides phase level details about power consumption, processing time and different hardware performance counters. Since the data is collected at different phases of the application, there are more than 4,000 phases across 19 different single, multi-threaded and graphics applications. This framework and data were used to design algorithms [22, 24, 45] that optimize runtime power, performance, and energy of heterogeneous target platform.

Target Platform Odroid XU3 (Section 3.6): The third step in developing a robust power optimization technique is choosing a correct target platform. Since the focus of this instrumentation framework is mobile heterogeneous architecture, Odroid XU3 is used as a target platform which is not only ARM based but also has eight multi-heterogeneous cores equally divided into big (A15) and little (A7) clusters. Odroid XU3 [27] is used to implement proposed instrumentation framework and then is also used to collect phase level performance data with the help of different hardware counters and sensors. This platform provides us with numerous cores and frequency knobs to enable optimization. This section also provides deep insight into Odroid XU3 in terms of its system and hardware features.

Using Instrumented Framework For Performance Optimization (Section 3.7): Final step is to make use of the developed framework and data collected to implement algorithms that enable dynamic control of the type of cores, number, and the frequencies of active cores in heterogeneous processors. This section provides a brief overview of the experiment methodology that can be used to integrate the instrumented framework with any of proposed optimization methodology to enable faster experimentation and testing. There are two major runtime optimization techniques that make use of proposed instrumentation methodology. The first work [22] enables run time optimal configuration selection on MpSoCs using traditional classifiers. The second work [45] provides optimization based on Oracle policy using Imitation Learning. **Chapter 4** presents results based on instrumented data which is used to train and test the published performance management framework.

Linux Perf VS PAPI (Section 3.4): Linux Perf [37] is a widely used profiling tool on Linux Platforms. Since the target platform also runs Linux operating system, it is important to justify why to make use of PAPI [49] and not Linux Perf. One of the key advantages of PAPI over Linux Perf is the ability to allow phase level instrumentation. With Linux Perf only time based sampling is possible, which is not the goal of this work. This section explores the benefits of using PAPI over Linux Perf for instrumentation.

3.2 Motivation

Most of the existing DVFS solutions around power and performance optimization rely on application level characterization. The fundamental problem with this approach is that not all part of the application requires same optimal configuration to run because an application is made of different phases and each phase has different optimal needs from the underlying architecture. Mobile platforms are equipped with a growing number of cores in MpSoCs and each architecture provide numerous configurations that can aide run time governor based optimization. The target platform used in this work is equipped with ARM big.LITTLE Samsung Exynos 5422 MpSoC which has four little A7 cores cluster and four big A15 cores cluster that can operate at 13 and 19 different frequencies, respectively [27]. Each of the A7 and A15 core clusters can scale different voltages by adjusting the frequencies. In all, there are total of $4\times5\times13\times19$ (4940) different frequency and core configurations. Running the platform on each of these configurations leads to a huge variation in power consumption and performance as shown in Figure 1.1. Moreover, any given applica-

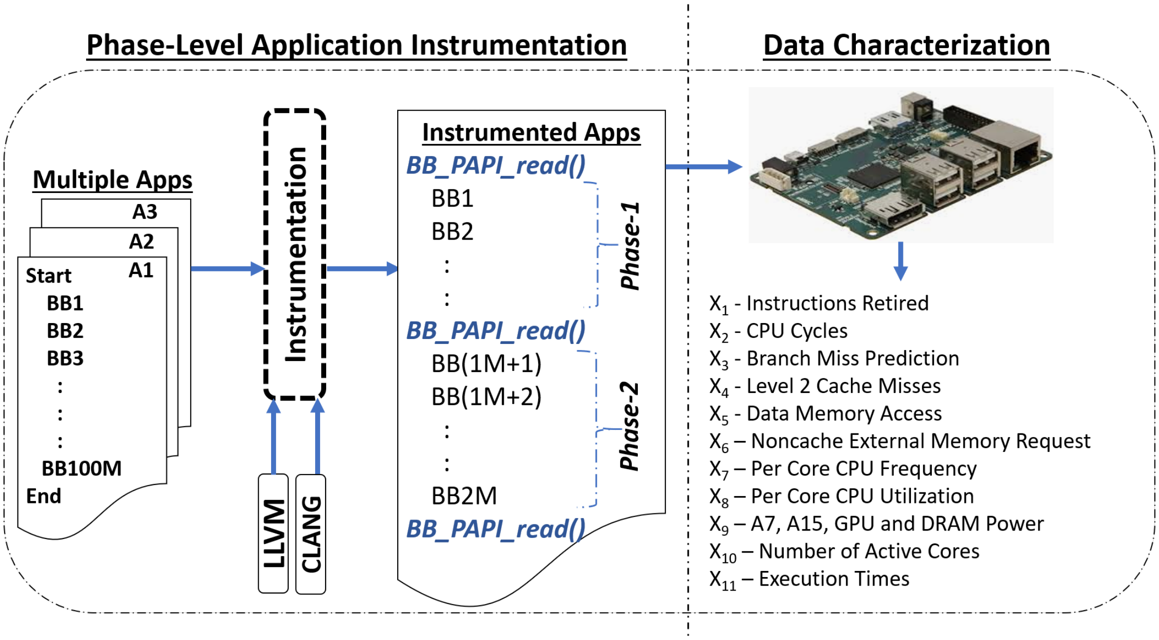

Phase-Level Application Instrumentation
Data Characterization
Multiple Apps
A3
A2
A1
Start
BB1
BB2
BB3
BB100M
End
Instrumentation
LLVM
CLANG
Instrumented Apps
BB_PAPI_read()
BB1
BB2
Phase-1
BB_PAPI_read()
BB(1M+1)
BB(1M+2)
BB2M
Phase-2
BB_PAPI_read()
X1 - Instructions Retired
X2 - CPU Cycles
X3 - Branch Miss Prediction
X4 - Level 2 Cache Misses
X5 - Data Memory Access
X6 – Noncache External Memory Request
X7 – Per Core CPU Frequency
X8 – Per Core CPU Utilization
X9 – A7, A15, GPU and DRAM Power
X10 – Number of Active Cores
X11 – Execution Times

Figure 3.1: Phase-Level application instrumentation and data characterization process.

tion consists of multiple workload phases [62]. For example, lower CPU frequencies may save power during a memory-intensive phase. In contrast, CPU-intensive phases with many active threads are likely to benefit more from higher frequencies and number of cores.

Therefore, different configurations may become optimal with respect to a given metric as the workload varies at runtime [10]. To make use of different phases of an application to enable runtime optimization, there is a need for a framework that provides platform runtime data for each of these phases. The work done in this thesis enables the development of such solution where any given application can be accurately profiled to find the phases and then instrument these phases to collect runtime phase wise characterization data which enables optimization algorithm development. Later sections in this chapter provide more details about the instrumentation process.

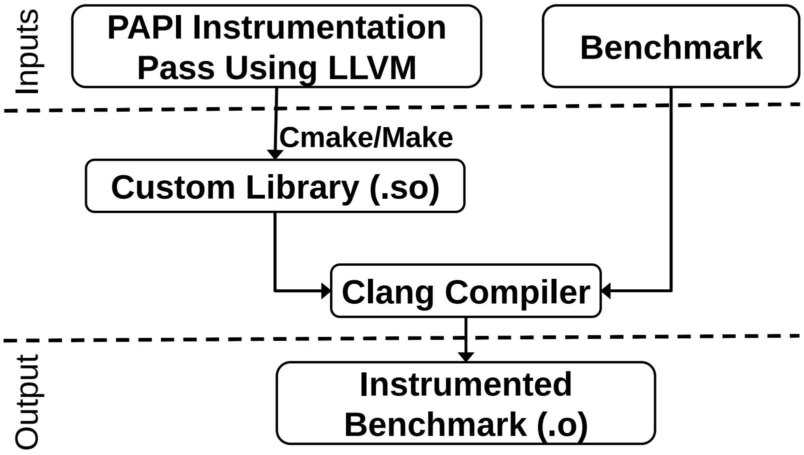


Figure 3.2: PAPI instrumentation overview.

3.3 Phase-Level Application Instrumentation

This section discusses in detail the phase level application instrumentation, terminologies around instrumentation and how it is implemented.

Defining Phase in an Application: Any application is made of different code blocks. Two major types of code blocks are: function blocks and basic blocks. Each of these blocks is responsible for executing a specific code block in the application code. A function block can have many basic blocks but vice versa is not true. Every basic block has a single entry and single exit point that and any given application can have millions of basic blocks. Performing instrumentation at a functional level will not provide details about important phases at basic block levels. Hence, this work focuses on using basic blocks for instrumentation. Application phases in this work, are also later defined as snippets.

Source-Level Instrumentation: In the software world, the instrumentation is a process of adding an extra piece of code without affecting the original execution flow of the application. In order to, achieve accurate phase-level instrumentation, source-level instrumentation is used rather than binary-level instrumentation. Using source-level instrumentation all the basic blocks in an application are found using basic blocks with the help of LLVM compiler infrastructure as shown in Figure 3.2. Using this information, accurate instrumentation of PAPI calls is achieved which enables data characterization.

PAPI: All the applications used in this work are instrumented with PAPI calls. It is important to understand what these PAPI calls are and what they do. Performance Application Programming Interface (PAPI) provides system independent access to hardware performance counters found on most of the processors. The target platform on which PAPI is running should have Performance Monitoring Unit (PMU) which provides cycle based performance details such as the number of instructions retired, cache misses, memory access, etc. This information is stored in specific registers that can then be read with help of software calls. This is where PAPI comes into use. By using the API calls in PAPI library, software calls can be made to read run time performance counter data. There are two types of performance counters also called as events. The first one is the a set of native events and then another is set of preset events. Preset events are more generic events that are defined and by default found on the majority of the architecture such as instructions. Native events are more target specific. It can be read with the help of PMU and platform perf driver by making calls to find which specific native events other than preset events are supported by the target platform. In this work, both preset and native events are used. Since our goal is to instrument the code that will run and collect data during the run time of the application, it is important to select set of those events which cover different CPU architecture details such as pipeline, cache, and memory. Hence, the performance counters instrumented in this work are chosen such that they provide details about different parts of the core architecture. Apart from performance counter events, different sensors were used to log data such as the total CPU and DRAM power consumption, core frequencies, core utilization, and also phase level execution time by leveraging PAPI. The system and application level parameters used in this work are listed in Table 3.1.

Finding Basic Blocks: As described in the previous few subsections, the phases are se-

Table 3.1: System and application level parameters used in this work.

Application Level Parameters	System Level Parameters
Instructions Retired	Per Core CPU Frequency
CPU Cycles	Per Core CPU Utilization
Branch Miss Prediction	little, big, GPU and DRAM Power Consumption
Level 2 Cache Misses	Number of Active Cores
Data Memory Access	Execution Time
Noncache External Memory Request	

quences of basic blocks. To instrument at basic blocks level, a process to find them is required. The instrumentation framework proposed achieves it by relying on LLVM and *clang* compiler. Instrumentation pass in LLVM identifies existing functions and basic blocks, and add new functions (PAPI APIs) for any application.

Instrumenting Application with PAPI: Figure 3.2 illustrates the process of instrumenting any benchmark with PAPI calls using LLVM and *clang* compiler. Using Cmake/Make utilities to compile the LLVM instrumentation pass to get a custom library object file. Finally, *clang* compiler is used to compile the benchmark with the custom library as an additional input. This generates an output object file that has PAPI instrumented at different basic blocks. This instrumentation process is independent of how the application code is written, as it relies specifically on analyzing the basic blocks, which are the building blocks of any application and a widely used syntax analysis terminology in the compiler domain. To accurately instrument applications with PAPI, LLVM compiler infrastructure is used as it has the functionality to analyze any given source code at different regularities, such as module level, function level, and basic block level [43]. LLVM treats any input source as a single block of module that can be broken down into functions. Each of these functions contains different basic blocks that subsequently contain assembly instructions. Instrumenting at the function level is too coarse, while instrumentation at the instruction level is too fine-grained. Therefore, utilizing LLVM with *clang* compiler [42] to analyze and instrument PAPI calls at critical basic blocks within an application to collect the hardware counters at runtime.

All the target applications are instrumented to divide into groups of basic blocks called snippets. This enables the collection of power and performance statistics of each snippet at runtime, as illustrated in Figure 3.1. For example, consider an application code with 100 million basic blocks (BB1 to BB100M) where each basic block is a sequence of instructions. The instrumentation in this example inserts special $\mathrm{BB\_PAPI\_read()}$ basic blocks that call the PAPI for reading hardware counters and system statistics every 1 million basic blocks. A pair of $\mathrm{BB\_PAPI\_read()}$ basic blocks create a boundary for different snippets of an application. Each snippet or a sequence of snippets may form distinct phases.

Instrumenting single-threaded workloads requires identifying the critical basic blocks and then adding simple PAPI calls. While instrumenting the multi-threaded benchmarks, tie each thread to its performance counter values. It is achieved with the help of PAPI, which provide specific calls to register threads that can maintain their counter data. Since multi-threaded workloads also have phases that only have single threads, it is ensured that instrumentation can capture such phases as well. At the time of logging the hardware counter values along with system performance, thread IDs and time-stamp of data collection is also written to the log file. This methodology allows the analysis of both single- and multi-threaded phases of any workload. In practice, inserting PAPI APIs are expected to introduce extra instructions as overhead. Therefore, it is ensured that the overhead introduced with these API calls is less than 1%. Overall, the process of instrumentation enables capturing the critical regions that provide useful information regarding different phases of an application running on any platform.

3.4 PAPI vs Linux Perf

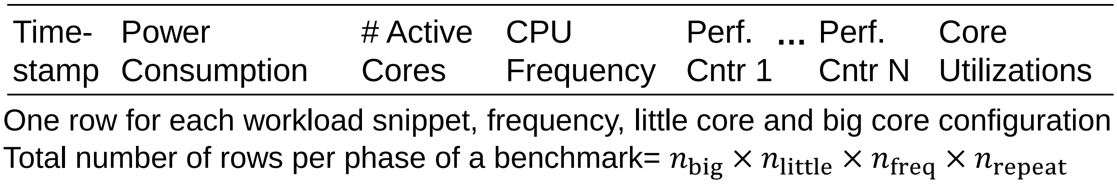
For many optimization based analysis using performance counter, Linux Perf is the most preferred tool. It can be used with any application during runtime to easily capture performance counter data and also logs it to a text file. There are many shortcomings of Linux Perf in comparison to PAPI. Linux Perf does not enable profiling at the basic block level and it is also not useful when it comes to getting data for every basic block or snippet. It is possible to log performance counter data every specific time interval using Linux Perf, however such data still does not provide which part of the application the data belonged to. Also since the goal of this work is to enable phase-level instrumentation rather than time based instrumentation, PAPI is preferred.

Since the goal of this work is to enable not only phase level instrumentation but also provide CPU specific data like power consumption, active cores, per core frequency and most important per core utilization. Such data is not easy to get using Linux Perf and also based on the preliminary analysis it adds approximately 5ms of additional runtime overhead. Another major benefit of using PAPI to enable instrumentation framework over Linux Perf is *portability*. Apart from ensuring that there is a perf driver and PAPI library is installed on the system, the proposed instrumented framework can be easily ported to a new system. With Linux Perf it is also very difficult to integrate any optimization framework that can take decision based on runtime data. Using PAPI this work can provide a framework that not only instruments application at the basic block level, but also allows algorithm integration for MpSoCs power and performance optimization.

3.5 Data Characterization Methodology

Once the benchmarks are instrumented with the PAPI, runtime instrumented data is collected for different frequency and core configurations. First, the platform is set to the

Table 3.2: Data format for each phase.



Time- stamp	Power Consumption	# Active Cores	CPU Frequency	Perf. Cntr 1	...	Perf. Cntr N	Core Utilizations

One row for each workload snippet, frequency, little core and big core configuration
Total number of rows per phase of a benchmark= $n_{\mathrm{big}} \times n_{\mathrm{little}} \times n_{\mathrm{freq}} \times n_{\mathrm{repeat}}$

highest frequency and core configuration, i.e., 2 GHz for the big cores with all eight cores active. Then, three iterations of each benchmark are executed at this frequency and core configuration. Next, by stepping down the frequency of the big core cluster while maintaining the number of active cores. The same process is repeating for each benchmark included in the study. After this, by reducing the frequency level by one, and the same process is repeated for all the supported frequency levels and the core configurations.

This selection includes all core configurations (4×4) from 1L+1B to 4L+4B. At least one little and one big core is included in order to maintain the heterogeneity of the system. Then frequency is sweeped uniformly from 0.6 GHz to 2 GHz in steps of 0.1 GHz for all 16 core configurations. Frequencies lower than 600 MHz are not included since they are rarely energy optimal. That is, the lowest power configuration in our experimental setup is {1L, 0.6 GHz, 1B, 0.6 GHz} and the highest performance configuration is {4L, 1.4 GHz, 4B, 2 GHz}. The entire application is run from start to end for all the selected configurations. Therefore, all the relevant phases are considered irrespective of the application. On profiling three iterations of 19 benchmarks for 256 different configurations lead to a total of 13,824 different benchmark runs. The system is re-booted before starting the data collection process for each benchmark to ensure consistency of the platform environment. Finally, all the data collected during the characterization data for each workload snippet following the format shown in Table 3.2.

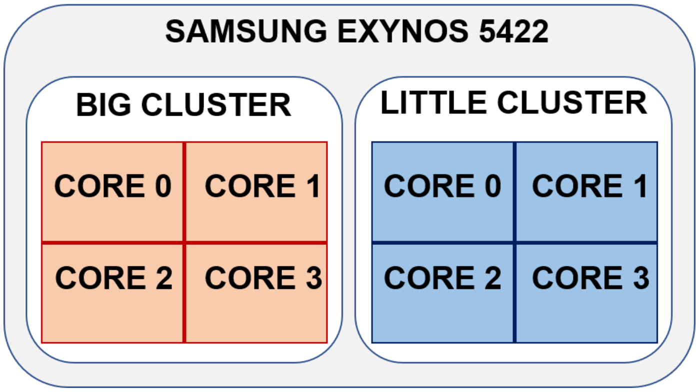


Figure 3.3: Odroid XU3 Exynos 5422 processor with big (A15) and little (A7) cores.

3.6 Odroid XU3

The experiments to collected the characterization data were performed on the Odroid XU3 platform running Ubuntu OS with kernel version 3.10 and 4.4 [27]. The platform is equipped with Samsung Exynos 5422 chip, which has four little (A7) cores and four big (A15) cores as shown in Figure 3.3. The little core frequency can vary from 0.2 GHz to 1.4 GHz and big core frequency can change from 0.2 GHz to 2 GHz in steps of 0.1 GHz. The platform supports per cluster DVFS, i.e., the cores within the same cluster have to run at the same frequency and voltage. Changing the CPU cluster frequencies and setting of the core online and offline are supported in the platform using the cpu-freq driver. The platform also provides INA231 current monitoring sensors [70] that report the power consumption for each CPU cluster, memory and GPU using the I2C driver. The sampling frequency of the current sensors is set to 5 ms to capture small transients in power consumption.

3.7 Using Instrumented Framework For Performance Optimization

The major goal of the instrumented benchmarks and the data collected on Odroid XU3 is to enable easy integration with any proposed power and performance methodology. Figure 3.4 shows how PAPI calls instrumented in the benchmarks can also be integrated with

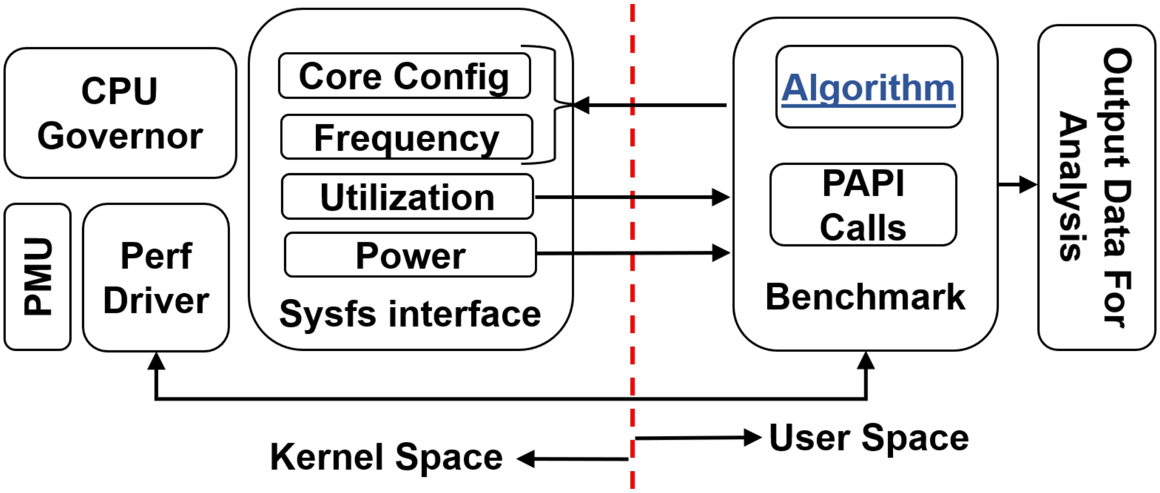


Figure 3.4: Power and performance optimization algorithm can be integrated with instrumentation framework for evaluating the algorithm.

any algorithm that proposes optimal running condition for different phases of the application. This implementation is divided into the kernel space and user space. The *kernel space* contains the Perf driver and the CPU governors with a sysfs interface [48]. The Perf driver is mainly responsible for communicating with the ARM performance monitoring unit (PMU) [14], which keeps track of different hardware and software counters, which is the key to the data collected from each of the applications in Table 3.3. As discussed earlier, it allows the PMU to capture the performance counters listed in Table 3.1. The framework also utilizes a custom CPU governor to capture per-core utilization through the sysfs interface.

The *userspace* contains the instrumented benchmarks with PAPI that query the perf driver for performance counters [49]. At runtime, the hardware counters and CPU utilization at each snippet of the application are used and can be fed to any of the optimization algorithms. During the runtime, any classifier based technique can find the optimal frequency and core configuration, and then assigns them to the cores using the sysfs interface. The instrumentation framework also logs time stamps, algorithm output, and input features

Table 3.3: List of applications instrumented from different bench marking suites.

MI-BENCH	**CortexSuite**	**PARSEC**	**Gaming**
Basicmath Large	Kmeans	Blackscholes 2 Threads	Tetris
Dijkstra	Spectral	Blackscholes 4 Threads	
FFT	Motion Estimation	Fluidanimate 2 Threads	
Patricia	PCA	Fluidanimate 4 Threads	
Qsort			
SHA			
Blowfish			
String Search			
ADPCM			
AES			

to a log file for debugging and offline analysis purposes.

Benchmarks: To validate the implementation, nineteen single- and multi-threaded benchmarks from PARSEC [9],MI-Bench [26], Cortex [69] suites and a standalone gaming application Tetris are used. Table 3.3 provides the list of all the applications instrumented from each of the benchmarking suites.

Usage With Default Governors: The Linux kernel implements many frequency governors that allow developers to optimize for a certain parameter. The *powersave* governor runs the application at the lowest frequency such that the power consumption is minimized. The *ondemand* governor is used to meet a user defined utilization threshold by changing the frequency [54]. The *interactive* governor is similar to the ondemand governor, except that it holds the frequency at a certain level for a fixed interval before making any changes. The instrumentation framework is designed to work with any of the governor using the methodology as shown in Figure 3.4. It is important to note that governor, except *ondemand* does not provide per core utilization, however the system around the instrumentation frameworks not only enables per core utilization log but it does so for *powersave* and *interactive* governor too.

Overhead Analysis: There is always a possibility that the addition of instrumented code to the vanilla benchmark will result in overhead. This is true for the proposed instrumentation framework too. However, the measured runtime overheads are almost negligible. The instrumentation overhead can be measured in terms of the percentage of the extra instructions added to the benchmarks to log the performance counter data using the PAPI. The baseline is the case when no APIs are inserted within the benchmark. As opposed to the baseline, the APIs in this approach have to be added in the source code from different workload snippets, as explained in Section 3.3. It is observed that a very low mean and median overheads of 1.0% and 0.2% across all the 19 different benchmarks as listed in Table 3.3. However, there may be an overhead from the integrated algorithm. Based on the usage in [22], the overhead of instrumentation framework along with runtime selection algorithm is 20s, whereas the minimum and mean execution time of the workload snippets are 2.1 ms and 22.6 ms, respectively. That is the runtime overhead of is less than 1% of the smallest snippet and less than 0.1% of the mean value of the execution time of all the snippets. Similarly, the instrumented framework in [45] measured the run-time overhead of both the instrumentation framework and policy for each snippet in the application set. The execution time of the policy ranges from 13μs to 200μs. This amounts to an overhead of 0.07% to 1% for an average snippet of length 20 ms. Both the approaches show that there is negligible overhead when the algorithm is integrated with the instrumented framework.

3.8 Conclusions

The instrumentation process and data collection methodology proposed in this work is unique and provides the ability to profile application at basic block level while the application is running on the target platform. This methodology has led to two published work [22, 45] that showcase how such unique set more than 4000 phases across 19 different set of application can enabled runtime optimization. This work not only proposed a novel instrumentation framework but also enabled runtime testing of power and performance optimization technique. The data collected at the phase level can also be used to propose many future novel runtime schemes. This data can also be critical in providing runtime online learning [6, 23] that can help optimize several unseen applications on MpSoC platform.

Chapter 4

POWER, PERFORMANCE AND ENERGY MANAGEMENT USING PHASE LEVEL INSTRUMENTED WORKLOADS AND FRAMEWORK

4.1 Overview

The instrumentation and data characterization methodology as shown in Figure 3.1 and the Figure 3.4 framework created around it is designed to easily integrate with any power, performance, and energy management optimization algorithm. Using the full setup as explained in **Chapter 3**, it is easy to integrate and test any optimization framework that enables runtime phase-level decision on heterogeneous architectures like Samsung Exynos 5422. This chapter will present details on how using the instrumentation framework and data, it enabled two published work. First work proposes optimal runtime phase-wise configuration classifier [22]. The second work uses data to use it for imitation learning [45] to better earlier phase level optimization technique.

This chapter is organized as follows:

Enabling Proposed Methodology (Section 4.2.1 and 4.3.1): This section briefly describes two different methodologies that enables runtime optimization on Odroid XU3. The data collected using the instrumented benchmarks as explained in Section 3.5 allows offline characterization to develop optimization technique. The structured data can also be used to either accurately train a classifier and built Oracle policies which outperform default governors.

Faster Experiments (Section 4.2.2 and Section 4.3.2): Apart from making use of data to enable runtime optimization algorithm development, the proposed framework also enable easy experimental validation. The proposed methodology shows how two different algorithms can plug into the instrumentation framework to quickly validate and evaluate the proposed methodologies. The same framework in future can be used to enable new novel online strategies on heterogeneous architectures that can provide optimal core configuration for unseen applications.

Improved Results (Section 4.2.1 and Section 4.3.1): Using the instrumented data to generate classifiers and oracle, both proposed solutions show improved results when compared with previous best methodology. The section presents the results showcasing how much performance per watt improvement is there compared to existing methodologies. Data is critical in providing a base that allowed detailed offline characterization without which such algorithm development is not possible.

Summary (Section 4.2.4 and Section 4.3.4): This chapter concludes by summarizing major takeaways in terms of instrumentation, data and how it acts as a catalyst in enabling solutions that are not unique but far better compared to existing runtime heterogeneous governors. In future, such big amount of data collected at all possible frequency and core steps will drive further optimization on heterogeneous architectures.

4.2 DyPO: Dynamic Pareto-Optimal Configuration Selection

DyPO [22] is one of the two runtime optimization techniques that made use of instrumentation data and framework in **Chapter 3**. This section provides a brief overview of how the phase-level data is used to propose a methodology to enable runtime optimal selection. Later sub-sections showcase the experiments carried out using the instrumentation framework and conclude with results and conclusion.

4.2.1 Proposed Methodology

In **Section 3.6** it is discussed that Exynos 5422 provides the ability to switch frequency and core configuration at two different clusters. With A7 (little cluster) and A15 (big cluster) more than 4000 optimal voltage and frequency configurations can be achieved. Each of these configuration settings provides varying optimal conditions. In DyPO, the set of all possible configurations is denoted by $\mathbf{C}$, and the configuration at time k with $c_k \in \mathbf{C}$. Each feasible configuration can be represented by $c_k = \{n_{L,k}, f_{L,k}, n_{B,k}, f_{B,k}\}$, where the elements represent the number of active little cores, the frequency of little cores, the number of active big cores, and the frequency of big cores, respectively. Similarly, the set of phases encountered during the lifetime of an application is denoted by $\mathbf{P}$, and the phase at time k with $p_k \in \mathbf{P}$. DyPO's optimization goal can be expressed as:

$$\begin{aligned} &\textbf{Find}\ \ f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C} \qquad (4.1)\\ &\qquad \textit{where } c_k^* \in \mathbf{C} \text{ is the optimal configuration}\\ &\qquad \textit{for workload phase } p_k \in \mathbf{P} \end{aligned}$$

Identifying the optimal configuration c_k^* at runtime for each phase p_k is a daunting task due to a large number of workloads and configurations. For example, the Basicmath application has three phases, and identifying the optimal configuration of each phase would mean searching through 4004^3 ($\approx 6 \times 10^{10}$) different possibilities for the entire application. Searching through this combinatorial space is intractable at runtime. Furthermore, the definition of the optimality may change over time depending on the application scenario. For example, minimizing energy consumption becomes a priority when the battery is running low. Hence, there is a strong need to dynamically identify the optimal configuration c_k^* for a given optimization objective at any point in time.

DyPO designs a classifier using this characterization data (Section 3.5) as shown in the top part of Figure 4.1. For example, consider two different phases, the first with 10K *LLC-misses* (high) and the second with 1K *LLC-misses* (low). Suppose that the characterization step reveals the optimal configurations as {2L, 1 GHz, 3B, 1 GHz } and {4L, 2 GHz, 4B, 2 GHz }, respectively. The classification step uses these data points to design a

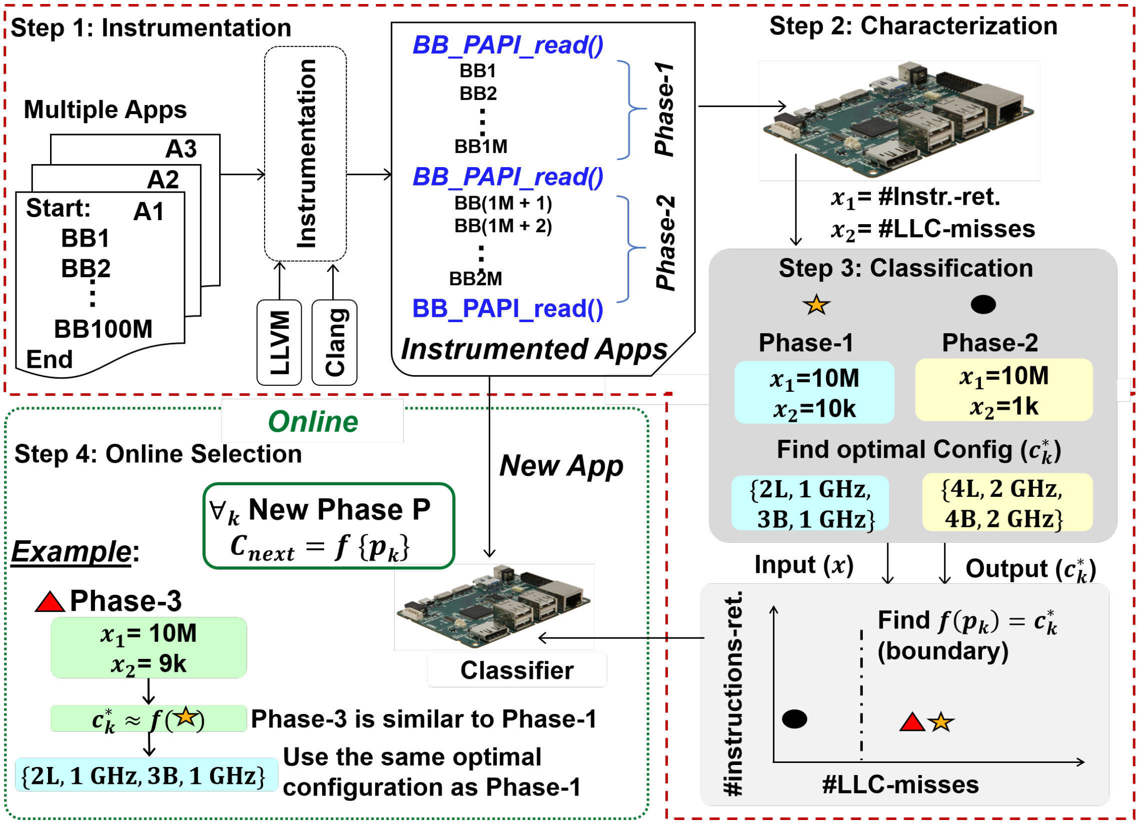


Figure 4.1: The outline of the DyPO [22] approach with an illustrative example. A block of instructions, such as a function call, makes up basic blocks. The instrumentation groups a sequence of basic blocks into distinct snippets. Finally, each snippet or a sequence of snippets may form workload phases.

classifier $f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C}$ that maps different snippets to the optimal configurations at runtime. The plot in the lower right corner of Figure 4.1 illustrates a potential classifier that can separate these two snippets. The classifier is then used online to determine the optimal configuration for any workload encountered at runtime. As an example, assume that the system encounters Phase-3, which has 9K *LLC-misses* and the similar number of *instruction-retired* with Phase 1 and Phase 2. Since Phase-3 is closer to Phase-1 characterized offline, the classifier will assign it the same optimal configuration of {2L, 1 GHz, 3B, 1 GHz }. While the illustrative example is simple, the real problem is multidimensional and far more challenging than creating simple visual boundaries between phases. DyPO aims to solve this problem by making use of the instrumentation and data collection methodology and then applying it using the classifier.

4.2.2 Experiments

DyPO experiments involved two-step process. The first process involves using the instrumented benchmarks to enable data collection by running eighteen single and multi-threaded benchmarks from PARSEC [9],MI-Bench [26], Cortex [69] and suites on Odroid XU3 platform. Figure 4.1 shows an experiment setup along with classifier that enabled DyPO optimization framework. The frequency on Odroid XU3 Exynos 5422 big.LITTLE is first set to the highest frequency and core configuration, i.e., 2 GHz for the big cores with all eight cores active. Then, three iterations of each benchmark at this frequency and core configuration. Next, stepping down the frequency of the big core cluster while maintaining the number of active cores. Repeating this process for each benchmark discussed earlier. After this, by reducing the frequency level by one, and repeating this process for all supported frequency levels and core configurations. Time spent on collecting data for 128 configurations on Odroid XU3 is typically about 1-2 hours per benchmark. Then sweeping the frequency uniformly from 0.6 GHz to 2 GHz in steps of 0.2 GHz for all 16 core configurations. Frequencies lower than 600 MHz are not included since they are rarely energy optimal. That is, the lowest power configuration in the experimental setup is 1L, 0.6 GHz, 1B, 0.6 GHz, and the highest performance configuration is 4L, 1.4 GHz, 4B,2 GHz. On profiling three iterations of 18 benchmarks for 128 different configurations lead to a total of 6,912 different benchmark runs. For each of the benchmark different phases of the applications are considered and data is collected for each of these phases because of the phase-level instrumentation. Each data phase of the benchmark collected is in the format as shown in Table 3.2.

4.2.3 Results

This section presents the validation of the DyPO classifier at runtime. Using the instrumentation framework as shown in Figure 3.4 at runtime, DyPO reads the hardware counters, power consumption, and utilization during each workload snippet as inputs to the classifier. Then, the classifier computes the probabilities of the optimal configurations. Finally, the configuration with the highest probability is assigned to the system for the next.

Figure 4.2 shows the comparison between offline characterized data for the entire application run at different frequency and core configurations (○), the Pareto-optimal points for power-execution time trade-off (♢), the Pareto-optimal frontier for energy-execution time trade-off (—), powersave governor ($+$), interactive governor ($*$), ondemand governor ($\times$), and the proposed DyPO-Energy approach ($\triangle$). Since these plots show energy and execution time trade-off, the operating points closer to the Pareto-optimal frontier and low ordinate are desirable. The data points plotted using the green markers (○) show the relative locations of the Pareto frontiers and the configuration space. This is useful in debugging and analyzing how different governor results get placed relative to these points. Figure 4.2(a) shows the results for the Basicmath application. The powersave governor lies to the extreme right of the plot at about 20 seconds execution time and consuming about 10 J of energy; this is expected as the goal of the powersave governor is to minimize power consumption. However, it does not minimize energy consumption. In contrast, the DyPO-Energy approach runs the application at the lowest energy point of the Pareto frontier at about 14 seconds of execution time and 8.7 J of energy consumption. It successfully achieves the energy minimization goal while also improving the execution time. Similarly, the DyPO-Energy approach leads to much lower energy consumption when compared with the interactive and ondemand governors. More precisely, the energy consumption is reduced by 42% (15 J to 8.7 J) and 46% (16 J to 8.7 J), respectively. This demonstrates

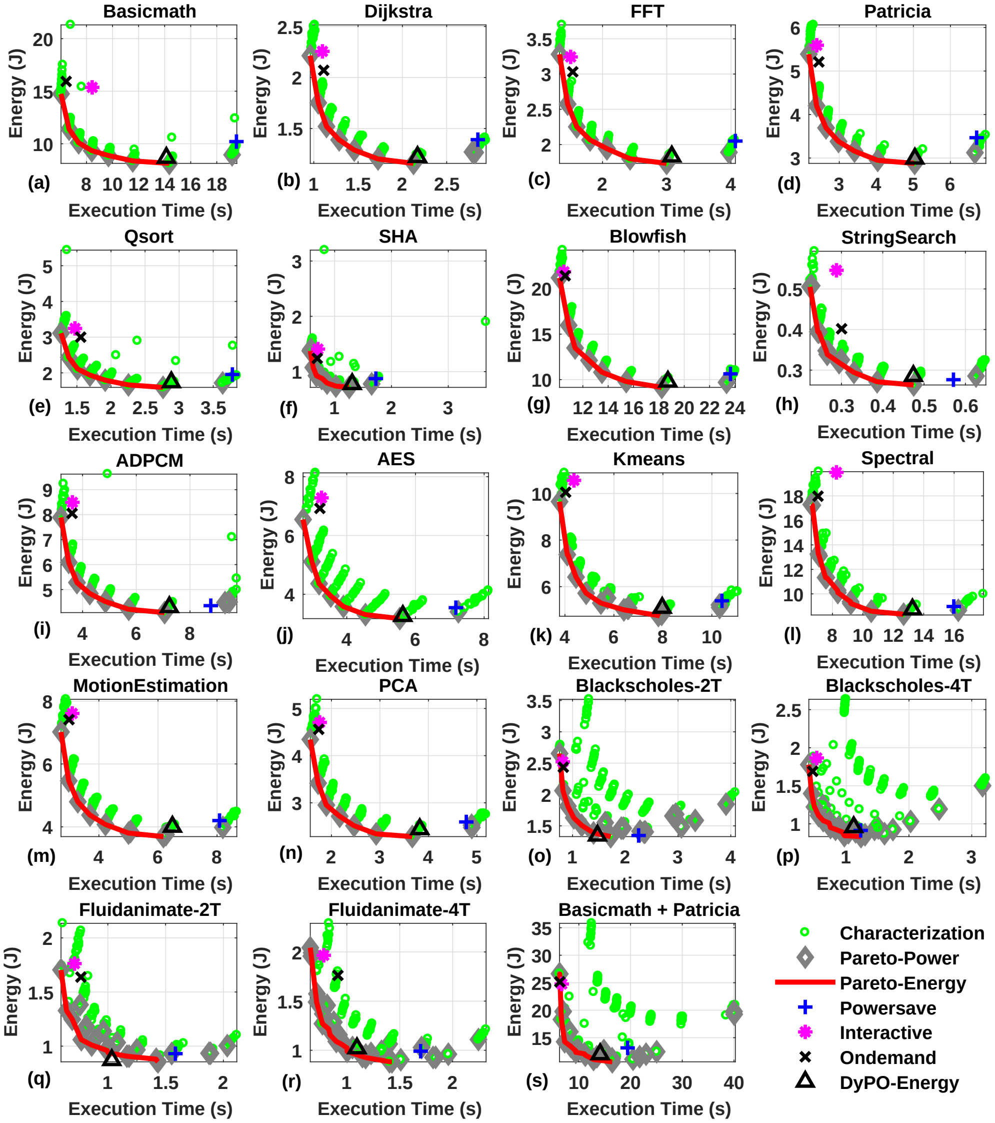


Figure 4.2: DyPO-Energy [22] approach compared with the default governors running on the platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively.

the effectiveness of the DyPO technique in optimizing energy consumption. More importantly, none of the three default governors in the system lie on the Pareto-optimal point.

In particular, the powersave and interactive governor are significantly off the Pareto curve. This is not desirable because there are other configurations in the system that could have achieved lower energy consumption for the same execution time. The rest of the plots in Figure 4.2(b-n) show the energy consumption and performance trade-off for 13 more single-threaded applications. As expected, the interactive and ondemand governors consume significantly more energy, since they are optimizing the system to meet a utilization target. The powersave governor, on the other hand, does a good job in reducing power consumption. However, it comes at the expense of performance and energy. In contrast, the results achieved by the proposed technique are always closest to the lowest point of the Pareto frontier for all applications.

Multi-threaded Applications: As the complexity of mobile apps increases, it is also important to analyze the behavior when running multi-threaded applications. Therefore, by analyzing their energy consumption and performance trade-off in Figure 4.2(o-r). In particular, Figure 4.2(q) shows the results obtained for the Fluidanimate application running with two threads. The DyPO-Energy approach lies below the Pareto-optimal curve, which means that the approach even outperformed the best case scenario of the characterization data, with a low energy consumption of 0.87 J and 1 second execution time. The lowest power configuration on the power and execution time Pareto curve (◇) leads to 2 seconds execution time. Moreover, it has substantially higher energy consumption compared to DyPO-Energy. This happens since the lowest power configuration utilizes a fewer number of cores, which has a very large penalty when there are more than one active threads. Similarly, the Blackscholes application running with two and four threads and Fluidanimate application with four threads show that the technique achieves lower energy than the default governors, as illustrated in Figures 4.2(o)(p)(r). In these workloads, the DyPO-Energy moves up on the Pareto-optimal curve towards higher performance. This happens since the active threads increase the utilization, which demands a larger frequency. However, the proposed technique still stays at the Pareto frontier, unlike the powersave, interactive and ondemand governors.

Concurrent Applications: The proposed runtime approach also works when multiple applications are running concurrently. More specifically, the instrumentation is specific to a particular foreground application. However, the classifiers operate on the performance counters, such as cache misses, non-cache external memory request, and the number of active cores listed in Table 3.1. Therefore, when other background applications are running, the load perceived by the governor changes. For example, the background applications can increase the CPU utilization, as well as hardware counters, such as LLC misses. Since the CPU utilization and hardware counters are inputs of the DyPO classifier, the proposed approach works with any number of applications and tasks running simultaneously with the foreground application. There were always hundreds of Linux OS background applications when experiments were performed. To demonstrate the operation with multiple applications more explicitly, experiment for simultaneously executing two applications, Basicmath (in foreground), and Patricia (in background). Figure 4.2(s) shows the results with this multiple application scenario. The proposed DyPO-Energy approach successfully minimizes the energy consumption compared to the default governors. More precisely, DyPO-Energy achieves 9% lower energy consumption, and at the same time, 27% faster execution time compared to the powersave governor. DyPO also observes 52% lower energy consumption than the ondemand and interactive governors, albeit with a significant increase in execution time. This is expected since DyPO-Energy minimizes the energy consumption, while ondemand and interactive governors aim for performance. Most importantly, the optimal energy consumption of BML and Patricia running together is 12 J. This is almost the same as the sum of the individual optimal energy consumptions of BML and Patricia from Figure 4.2(a) and (d) equal to 11.7 J (sum of 8.7 J and 3 J). This further corroborates the claim that multiple applications can be optimized by using the DyPO-

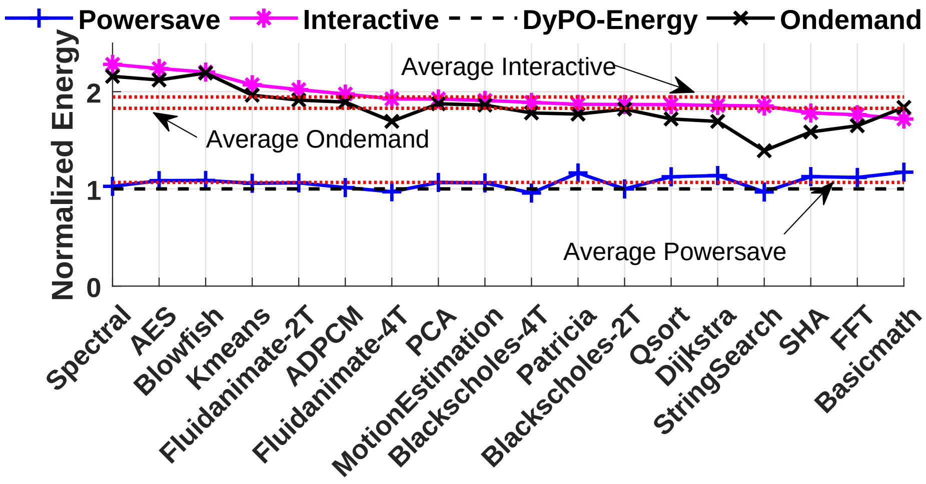


Figure 4.3: DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized energy consumption [22]

Energy approach effectively.

Comparison With Default Governors: This section summarizes the advantages of the proposed methodology in comparison to the default governors for each benchmark. To this end, by normalizing energy consumption, power consumption, execution time and PPW obtained for each governor with DyPO-Energy results. For example, Figure 4.3 shows the normalized energy consumption of all the benchmarks compared with the interactive, ondemand and powersave governors. It is observed that the energy consumption reduces by 49% and 45% compared to the interactive and ondemand governor, respectively. For the interactive governor, even the smallest energy savings obtained by DyPO-Energy for the Basicmath application is 41%. The energy consumption achieved by the powersave governor is slightly more than 6% of the energy consumed by DyPO-Energy. The power consumed by the interactive and ondemand governors is about 3.5× that of the DyPO-Energy, as shown in Figure 4.4, while the power consumed by the powersave governor is about 23% lower. Note that compared to the powersave governor, DyPO-Energy provides both energy savings and higher performance. When compared to the ondemand and inter-

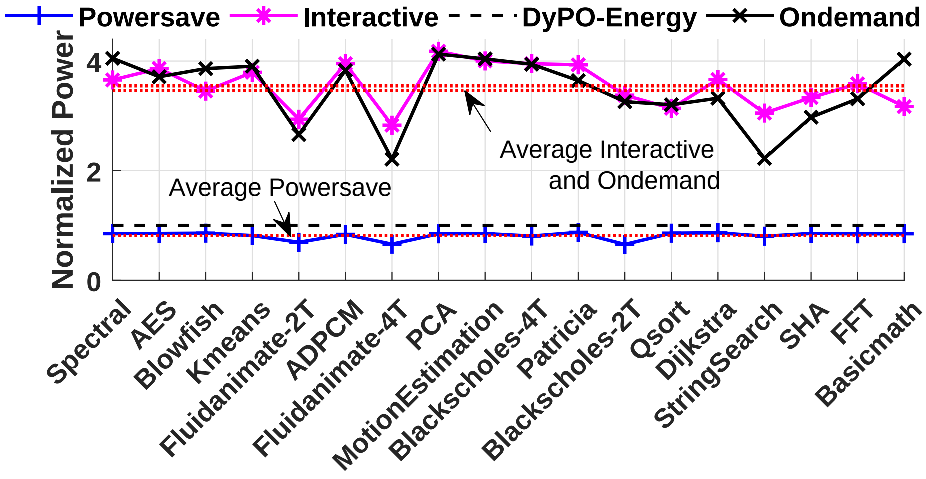


Figure 4.4: DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized power consumption [22].

active governors DyPO-Energy obtains substantial reductions in energy consumption albeit with lower performance, as shown in Figure 4.3. This is expected because the ondemand and interactive governors are designed for performance, not energy efficiency.

Comparison with Aalsaud et al. [1]: This section presents comparison of DyPO-Energy against a state-of-the-art approach proposed by Aalsaud et al. [1]. They use power and performance (IPC: Instructions/Cycle) models that are linear functions of the number of little cores, big cores and one bias term. Figure 4.5 shows the PPW obtained by the DyPO-Energy, Aalsaud-offline and Aalsaud-ADA approaches normalized to the PPW obtained by running the ondemand governor. On average, the DyPO-Energy, Aalsaud-offline and Aalsaud-ADA provide 81%, 46% and 18% gain in PPW compared to the ondemand governor. Therefore, the DyPO-Energy approach shows 55% and 25% improvement in PPW compared to the Aalsaud-offline and Aalsaud-ADA approaches, respectively. Note that for applications Blackscholes-2T and String-Search, both Aalsaud-ADA and Aalsaud-offline perform worse than the ondemand governor. This is because for the String-Search application, the Aalsaud-offline approach used the configuration with a frequency of 1.2 GHz, and

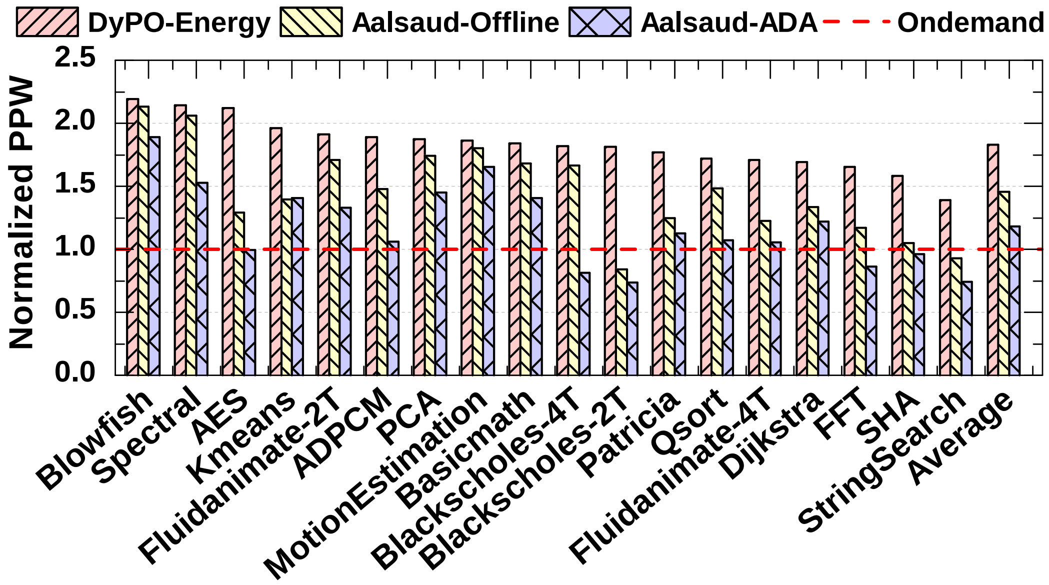


Figure 4.5: Comparison of the normalized PPW obtained using DyPO-Energy [22] approach and Aalsaud et al. [1].

four little and big cores. This wastes the extra energy headroom, whereas the ondemand governor utilizes it by keeping the frequency below 1 GHz. Similar behavior is observed for the Blackscholes-2T application. In contrast, DyPO-Energy provides substantial gains in PPW compared to the approaches in Aalsaud et al. [1] and to the ondemand governor for all the benchmarks.

4.2.4 Summary

Continued demand for performance led to powerful mobile platforms with heterogeneous multiprocessor system on chips. These platforms provide many voltage-frequency levels and active core configurations that can be chosen at runtime. DyPO presented a novel methodology that finds the Pareto-optimal configurations at runtime as a function of the workload. The methodology consists of a combination of offline characterization and runtime classification. Using phase-level offline characterization for several benchmarks and then using classifiers that map the characterized data to the Pareto-optimal configuration are learned offline using multinomial logistic regression. The classifiers are used at runtime to select the optimal configuration concerning a specific metric, such as energy consumption. Experiments show an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively.

4.3 Dynamic Resource Management Using Imitation Learning

DyPO presented a novel runtime optimal configuration selection classifier that can achieve better efficiency over existing state-of-the art governors using the instrumented and data collection framework. Using the same data collection methodology discussed work done in [45] presents a novel and practical imitation learning scheme for [39, 59, 61, 66] dynamic resource management of heterogeneous mobile platforms. The technique controls the type (Big/Little), number, and the frequencies of active cores. IL is a promising approach since it enables us to automatically transform an optimal offline solution into an efficient online policy. This work shows how phase-level instrumented data can be used to construct a near-optimal Oracle policy *offline* by exploiting characterization data for target applications available at design time. Then using advanced IL algorithms on characterized data one can design low-overhead control policies that imitate the Oracle policy and generalize beyond the applications used for training. This implementation and optimization proof is done on the exact same platform as DyPO. DyPO uses simpler logistic regression classifier while the proposed approach in this work used regression trees (RTs), thus does not limit the number of configurations available at runtime.

4.3.1 Proposed Methodology

The state-of-the-art mobile platforms similar to Odroid XU3 and Exynos 5422, can control the number of active cores and their frequencies at runtime periodically with 50 to 100 ms intervals as the workloads change dynamically [54]. Consider a heterogeneous

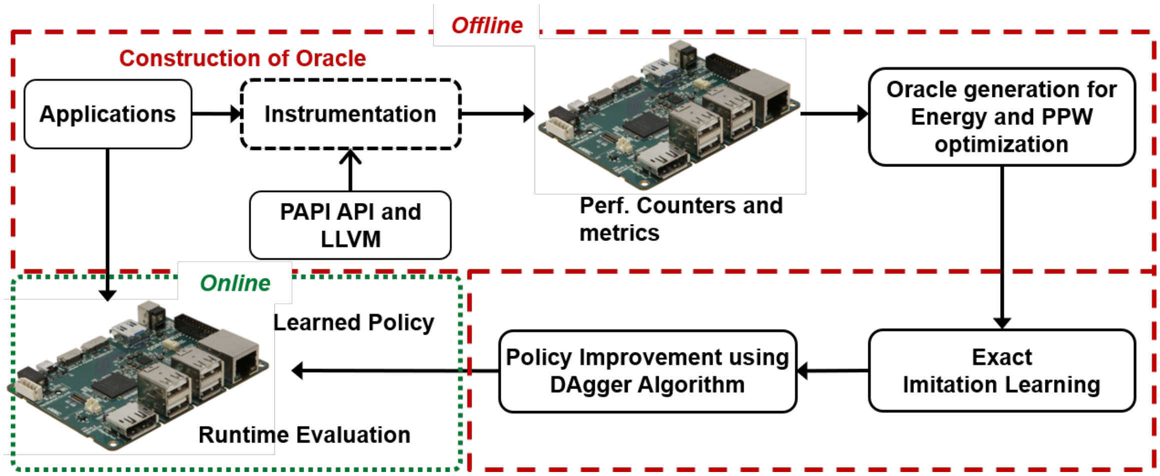


Figure 4.6: Overview of the proposed IL based dynamic resource management methodology [45]

mobile platform with n cores of k types. Without loss of generality, let us assume that one can control the platform at runtime by specifying the number of active cores for different types $(n_1, n_2, \cdots, n_k)$ and their corresponding frequencies $(f_1, f_2, \cdots, f_k)$. The set of supported configurations $\mathcal{C} = \{C_1, C_2, \ldots, C_M\}$, specify the number of active cores and their frequencies. For example, the hardware platform used in experiments has four Big (B) and four Little (L) cores. Thus, each configuration C_i for $1 \leq i \leq M$ in this platform is a 4-tuple $(n_{iB}, n_{iL}, f_{iB}, f_{iL})$ denoting the number of active Big cores, number of active Little cores, frequency of Big cores, and frequency of Little cores, respectively.

A given policy π maps the current system state to a candidate control configuration $C_i \in \mathcal{C}$ at each control epoch. Using the set of target applications $\mathcal{T}$ the goal is to create the best policy that maximizes the specified objective (e.g., performance-per-watt) on the given target applications that also generalizes to new unseen applications. This problem is solved using the instrumentation framework and characterized data to develop an IL-based methodology, as outlined in Figure 4.6. The proposed methodology [45] presents an efficient mechanism to construct an Oracle policy π^* that drives the overall learning process presented.

4.3.2 Experiments

The instrumentation, data collection and experiment methodology in this work is similar to DyPO except that an additional gaming benchmark is used and granularity of frequency increased. Since the framework used by in this work is same as that proposed in Chapter 3, it uses the same Odroid-XU3 board [27] running Ubuntu OS on Samsung Exynos 5422 SoC that integrates four A15 (Big) and four A7 (Little) cores. Since it supports heterogeneous multiprocessing, hence allows tasks to be scheduled to any of the eight cores. Evaluation of the proposed IL approach is proved using 19 application workloads from PARSEC with 'simsmall' inputs [9], MiBench [26], CortexSuite [69] benchmark suites and Tetris, which is a gaming application.

The major difference in terms of characterized data in this work compared to DyPO is the frequency step for each core and frequency configuration. In DyPO, the frequencies for which data collected on Big and Little cores were from 200 MHz–2.0 GHz and 200 MHz–1.4 GHz with a step of 200 MHz respectively. However, in this proposed Oracle based policy the frequency range is the same. However, the frequency granularity is 100 MHz, which doubles the amount of data available to make the policy more accurate. This vast amount of new data collection was possible due to the ease of using the instrumented framework created and described in Chapter 3.

4.3.3 Results

Application-Specific vs. Global Policy: Existing IL approaches on homogeneous many-core systems [39] employ an application based optimized policy. However, application-specific optimization is not practical since many applications are unknown, or may not even exist, at design time. Since controllers optimized for a specific application can be considered as a best-case solution, proposed IL policty [45] starts with comparing global

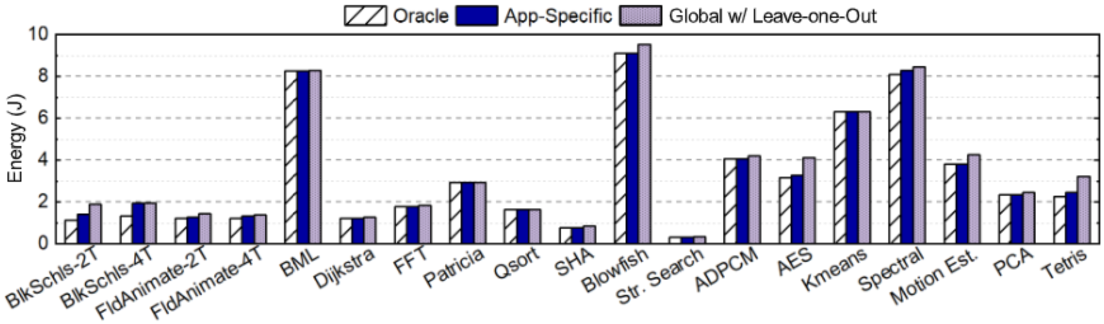


Figure 4.7: Comparison of energy between the Oracle, App-Specific and global policies using Oracle minimizing energy without timing constraints [45].

IL-based policy against application-specific policies. By training IL-based policy [45] for each application separately to obtain 19 app-specific policies. Then, using leave-one-out cross-validation and Oracle policies for all applications at once to generate a single global policy. That is, the test application is not included in the training of the global policy. Figure 4.7 compares IL global policy against the app-specific policies in terms of their total energy consumption. In this case, both global and application specific policies are trained with Oracle minimizing energy. The global policy performs very similarly (in terms of energy consumed) to app-specific policies which are not practical. The largest difference in energy between App-Specific and Global policy is observed for the Tetris application which is 30%. On an average, the energy consumption of the applications obtained by our global policy is within 6% of the app-specific policies and within 9% of the Oracle policies. In summary, the proposed IL methodology is able to produce a practical global policy which has comparable performance to app-specific policies while generalizing to unseen applications.

Comparison with State-of-the-Art Governors: Commercial devices are shipped with power and performance management governors like `performance`, `ondemand`, `interactive`, and `powersave`. Therefore, it is critical to compare the proposed IL approach with these default governors. It is achieved by running each application using all the governors. During this evaluation, characterization of performance, power consumption, and PPW of all the applications similar to earlier experiments are done. To provide a holistic view, comparison of these governors in terms of both performance and PPW is also presented.The performance governor operates close to the highest operating states whenever the SoC is actively running the applications. Therefore, normalize the execution time and PPW obtained for each application to the corresponding values given by the performance governor. Hence, the reference in the red square (□) marker in Figure 4.8 corresponds to the performance governor. Ideally, the results should be in the ideal region (marked in the dashed circle), where execution time is minimized, and performance is maximized. The results of the interactive and ondemand governors are shown with $\star$ and $\triangle$ markers, respectively. Each marker shows the normalized execution time (x-axis) and the percentage improvement in PPW (y-axis) in comparison to the performance governor. Both interactive and ondemand governors have low execution time but poor PPW, since they tend to ramp up the frequency whenever the cores are highly utilized. In contrast, the powersave governor, shown in blue circle markers (○) achieve on an average 92% higher PPW than the performance governor. However, this comes at the cost of more than 2.5× higher execution time.

The proposed global IL policy with PPW Oracle ($\Diamond$) *dominates the powersave governor* for all data points both in terms of PPW and execution time. The policy achieves on an average 15% boost in PPW combined with a 25% reduction in execution time. For Tetris application, the IL policy achieves 10% improvement in PPW with a 31% reduction in execution time when compared with powersave governor. While running Kmeans and BML applications concurrently, the IL policy achieves 150% improvement in PPW compared to the performance governor. At the same time, it reduces execution time, by 27% and consumes 5% less energy in comparison to powersave governor. The PPW improvement of IL

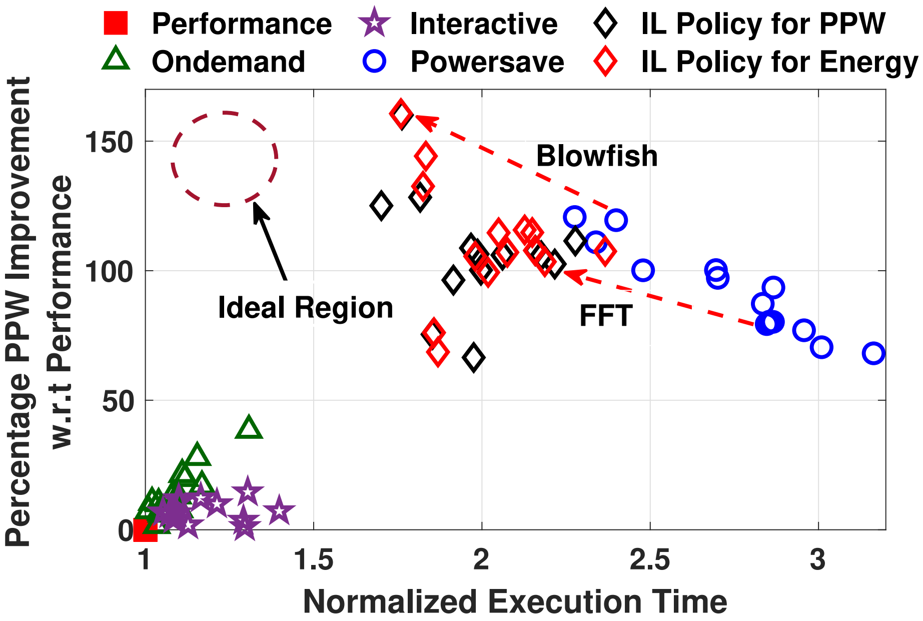


Figure 4.8: Comparison of the IL policy with default governors on Odroid-XU3. Markers represent different applications [45].

policy (with PPW Oracle) with respect to the performance governor ranges from 75% to 150%. Similarly, IL policy achieves significant PPW improvements when compared to the interactive and ondemand governors. The corresponding penalty in execution time is much smaller than that of the powersave governor.

The proposed methodology is also compared with the existing governors in the platform with the IL Policy that minimizes the energy consumption using red (◊) in Figure 4.8. It is observed that improvements in both PPW and execution time when compared to the powersave governor, which aims to minimize power consumption. The improvements for Basicmath and Blowfish applications are highlighted using red dashed arrows in Figure 4.8. The PPW improvement of the IL policy range from 101% to 159% with respect to the performance governor. For the gaming application Tetris, this IL policy consumes 39% less energy than powersave governor while having a 9% improvement in execution time. At the same time Tetris shows 152% improvement in PPW in comparison to the performance governor. Furthermore, while running Kmeans and BML applications concurrently, the policy achieves 5% more PPW than powersave governors with 36% improvement in execution time. At the same time, it achieves 117% increase in PPW than performance governor while running Kmeans and BML applications concurrently. The proposed IL policy for energy also achieves significant improvements when compared to the interactive and ondemand governors. It is also observed that the IL policy for energy minimization has a higher PPW than the IL policy for PPW maximization, but at a higher execution time penalty. The IL policy for PPW, on the other hand, achieves a lower penalty for execution time while sacrificing the improvement in PPW.

In summary, IL policies provide a significant step towards the ideal corner. They can replace the powersave governor and can be used in conjunction with the performance-oriented governors whenever the optimization goal is to maximize the PPW or minimize energy consumption.

Comparison with DyPO (Section 4.2): DyPO is a recent approach proposed in [22] for Pareto-optimal configuration selection in heterogeneous processors. The DyPO algorithm first identifies the Pareto-optimal configurations for a given performance metric and designs a classifier to choose the optimal configurations at runtime. This work also compares IL policies for PPW and energy with the DyPO-Energy algorithm in [22]. For all applications, the proposed IL policy has less energy consumption than DyPO. For BML, the energy consumption with the IL policy is 22% less than DyPO. For multi-threaded applications also, the IL policy shows improvement over DyPO. For example, 4-threaded Blackscholes application consumes 20% less energy when it is executed with IL policy than DyPO. Moreover, the proposed IL policy consumes 18% less energy than DyPO when Kmeans and BML applications run concurrently. On average, the proposed IL policy consumes 10% less energy than DyPO for the applications shown in the figure. Table 4.1 summarizes the average improvement in PPW, energy and execution time obtained by respective IL policies. It is

Table 4.1: Improvement w.r.t DyPO [22]

	Percentage Improvement		
Oracle type	**PPW (GIPS/W)**	**Energy (J)**	**Exe. time (s)**
Maximizing PPW	6.3%	4.7%	9.2%
Minimizing Energy	10.7%	9.3%	5.1%

noted that the proposed IL policy has 6.3%, 4.7% and 9.2% improvement in PPW, energy and execution time respectively with respect to DyPO when the objective of the Oracle is to maximize PPW. On the other hand, IL policy has 10.7%, 9.3% and 5.1% improvement in PPW, energy and execution time respectively with respect to DyPO when the objective of the Oracle is to minimize the energy consumed. The improvement over DyPO can be attributed to two primary reasons. First, DyPO limits the number of configurations available at runtime by pruning the configurations using the k-means clustering algorithms. As a result, it is unable to use the full breadth of configurations available in the platform. In contrast, the proposed approach has a much larger range of configurations to choose from at runtime. As a result of this, it can choose configurations that have better energy or PPW, depending on the optimization objective of the policy. Secondly, DyPO uses a much simpler logistic regression classifier while the proposed approach uses regression trees (RTs). RTs can approximate the nonlinear behavior of the policy better in comparison to logistic regression. In summary, the proposed IL-based policy can improve energy, PPW and execution time as well when compared to the DyPO approach.

Implementation Overhead: In this section, the evaluation of the implementation overhead of the proposed approach on the Odroid-XU3 platform [27]. To this end, the proposed imitation learning policies are implemented as user space governors. Measurement of the run-time overhead of policy for each snippet in the application set. The execution time

Table 4.2: Q-Table size for different number of bins.

Bins	**Features**	**State-Action Pairs**	**Accuracy (%)**
4	10	$10^6 \times 4940$	92.3
6	10	$6 \times 10^7 \times 4940$	95.1

of policy ranges from 13μs to 200μs. This amounts to an overhead of 0.07% to 1% for an average snippet of length 20 ms. The proposed approach also has a negligible storage overhead as it only has to store the comparison operators of the if-then rules corresponding to regression tree-based policy. Specifically, the regression trees implemented on the board have a maximum of 512 non-leaf nodes. This leads to a storage overhead of 2 kB for each RT. In contrast, RL approaches require the storage of a Q-table with one entry for each state-action pair defined by the features in Table 1 and available control actions (4940). For example, with four bins and ten features, policies achieve an average prediction accuracy of 92.3%, as shown in Table 4. Increasing the number of bins to 6 improves the accuracy to 95.1%. However, with 6 bins, the Q-table size grows to $6 \times 10^7 \times 4940$ entries, which is infeasible on most mobile platforms including ours.

4.3.4 Summary

Managing mobile platforms at runtime is challenging due to increasing heterogeneity, the number of processors, the large space of control actions, and exponential growth in applications. Existing governors on commercial devices employ simple heuristics based on system utilization, which leads to sub-optimal performance. IL presented a practical approach for dynamic management of mobile processors using the framework of IL. It propose construction of an Oracle policy to maximize PPW for a set of applications. Using this Oracle, runtime policies can be constructed that apply to a broad range of application workloads. Experiments on a commercial mobile platform show 101% improvement in

PPW on an average while running nineteen commonly employed benchmarks.

4.4 Conclusions

The two policies discussed in this chapter showcase the importance of accurately characterized data on heterogeneous architecture. Characterized data enables offline policies that enable runtime optimization. Both the proposed works [22][45] are enabled by the instrumentation framework discussed in Chapter 3. Phase-level instrumentation and characterized data enabled two different types of policies, with second one [45] bettering the other one [22] showcase that more such policies can be proposed using the same framework. By making use of characterization data as discussed in Chapter 3, both the policies can be further enhanced to achieve online training which will provide more advanced online optimal core frequency configuration policies.

Chapter 5

EXPLORATION OF MANY CORE PERFORMANCE ORIENTED HETEROGENEOUS ARCHITECTURE

5.1 Overview

Many core architectures are crucial in providing faster solution for workloads involving large amount of data processing. Graphics Processing Unit (GPU) is widely used for deep learning training which often involves thousands of gigabytes of data processing. Recently, Intel Xeon Phi code name Knights Landing (KNL) [65] have gained popularity which targets high performance computing including training neural networks using deep learning framework. Xeon Phi is also a heterogeneous architecture as each tile has multiple central processing units and quadruple vector processing units. These two different types of processing units allow performance gain and instruction set optimization. This works showcases architectural details of Intel Xeon Phi architecture and provides detailed exploratory analysis by running neural networks using deep learning framework Caffe [33] to train ImageNet data set [16]. This has led to work getting published [11].

Rest of this chapter is organized as follows:

Intel Xeon Phi (Knights Landings) (Section 5.3): To run deep learning framework and train networks on an open data set, it is crucial to understand the processor architecture on which training will occur. This section provides an overview of Intel Xeon Phi architecture by focusing on core, frequency and memory organization. It also presents details on the Network-On-Chip (NoC) that provides packet delivery from the core to memory and back to the core using YX mesh routing [31, 34, 44, 46, 65].

Memory and Cluster Modes (Section 5.3.2 and Section 5.3.3): Unique feature of Xeon

Phi is its ability to run at different memory and cluster modes. Xeon Phi's ability to provide a different combination of memory and cluster mode helps by reducing memory misses, eventually leading to reduced core-to-core communication time. These two features also provide the ability to run any application including training neural network faster. These sections provide more details on memory and cluster modes supported, and also show how different combination can allow performance gain.

Thread Modes (Section 5.3.4): When using GPU for running any type of kernel to train deep networks, threads within the software kernel play an important role. Just having a large number of threads is not important, it is also crucial to understand how to map these threads for given workloads such that there is less memory contention. Xeon Phi provides enough resources in terms of cores and memory to allow a large number of threads to run which eventually provides speedup. Xeon Phi relies on OpenMP for thread management. This section provides a detailed explanation on how thread management in Xeon Phi can be used and how different number of cores can provide enough resources to each of these threads. This section also provides details on the occurrence of thread bottleneck in Xeon Phi when the total number of threads running on a different number of cores leads to memory contention, eventually leading to performance degradation.

Deep Learning Framework, Networks And Data (Section 5.4): Over the last decade deep learning has attracted many researchers. This has enabled development of deep learning frameworks that allow not only usage of neural networks but also help enhance/propose networks that can provide much better accuracy using the vast amount of available dataset [16]. Deep learning framework has three major components - Frameworks that enables training, Networks that learns the data features and the Data itself. This section provides extensive detail of all these three important components of the many core Xeon Phi architecture.

Instrumentation, Experiments And Results (Section 5.5 and Section 5.6): This work carried out a detailed exploratory analysis on Xeon Phi and Deep Learning Framework on Developer Access Program [57] platform. With the help of performance counters and sensors on the system, deep insight into the architecture and where the bottlenecks are found. System tools like Linux Perf [37] were instrumented to not only log performance related data but also power consumption and execution time every specific interval.

5.2 Motivation

Deep learning has attracted a lot of research and many papers have been published around it [77]. This has led to the development of deep learning framework, learning networks and open data set. Data is crucial for deep learning research as the majority of the work is to train set of images to ensure that the network designed learns the features such that it can be applied to un-seen data to improve accuracy. ImageNet [16] solved the majority of the problem when it came to making large amount of categorized data available. By introducing different types of challenges ImageNet enabled creation of different networks [40] [28] [63] [67] that provide better accuracy using deep convolutional neural networks. To make it more easy for new researchers to evaluate and propose training networks, deep learning frameworks like BVLC Caffe [33] were developed.

These three components - data, network, and learning framework - led to research advancement in artificial intelligence (AI) and machine learning (ML). However, another important reason for AI and ML advancement is the processor architecture technology. GPUs with numerous nodes can provide performance which enables faster analysis, debugging and development of neural networks. Lately, there has been the need to also look into whether these architectures can be made more efficient [65] apart from being just powerful. Comparative study [68] has shown that GPUs are not the only architectures that should be used when it comes to training and testing network with huge amount of data. It also depends on the domain for which the training is being carried out.

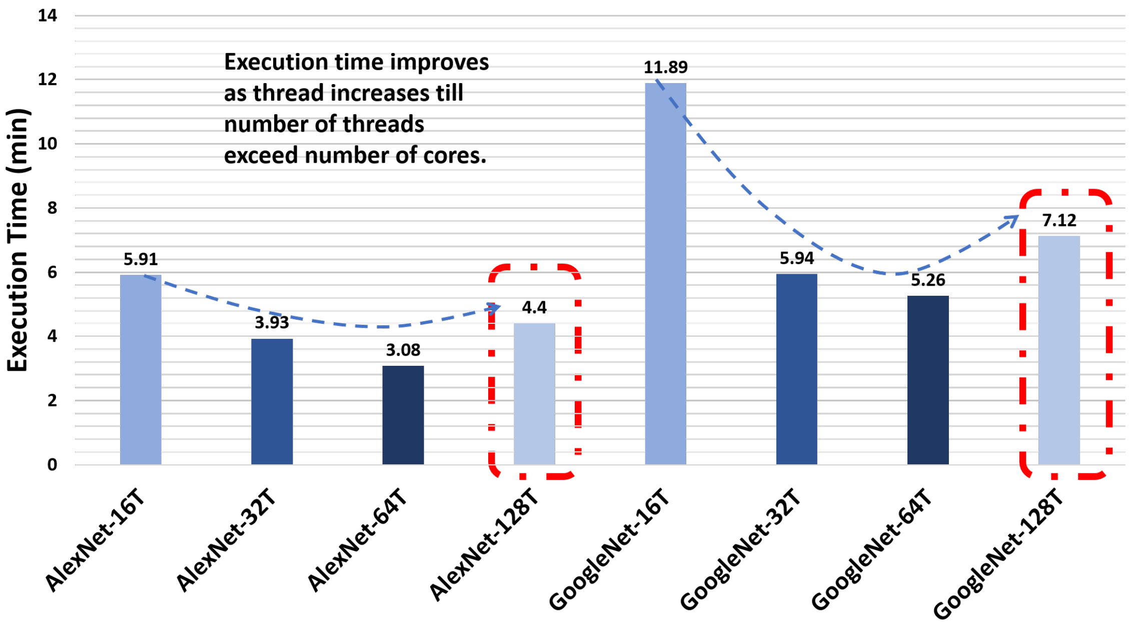


Figure 5.1: Running two types of networks on Caffe with increasing number of threads shows that Xeon Phi does provide speedup. Both AlexNet and GoogleNet ran for 100 iterations with a batch size of 256. Speedup does decrease when the number of threads start sharing resources. During 128 thread more than one thread is sharing same L2 cache within a single tile, thus leading to context switching and performance degradation.

Figure 5.1 shows how running two different types of deep CNN, AlexNet [40] and GoogleNet [67], with All-to-All cluster mode and Flat memory mode on Xeon Phi with an increasing number of threads (T in figure stands for thread) allows performance gain. However, from same figure one can also see that performance gain strongly correlates with the number of threads used to run during training. Since Xeon Phi is equipped with 64 physical cores with 2 cores sharing same 1 MB L2 cache, running more than 64T means resource sharing among threads. This eventually leads to cache thrashing and performance degrades during 128T as two threads are fighting for the same resources. Intelligent scheduling of each of the training task can boost training speed. Xeon Phi is one such processor architecture that provides the ability to train and test networks on a huge amount of raw data using

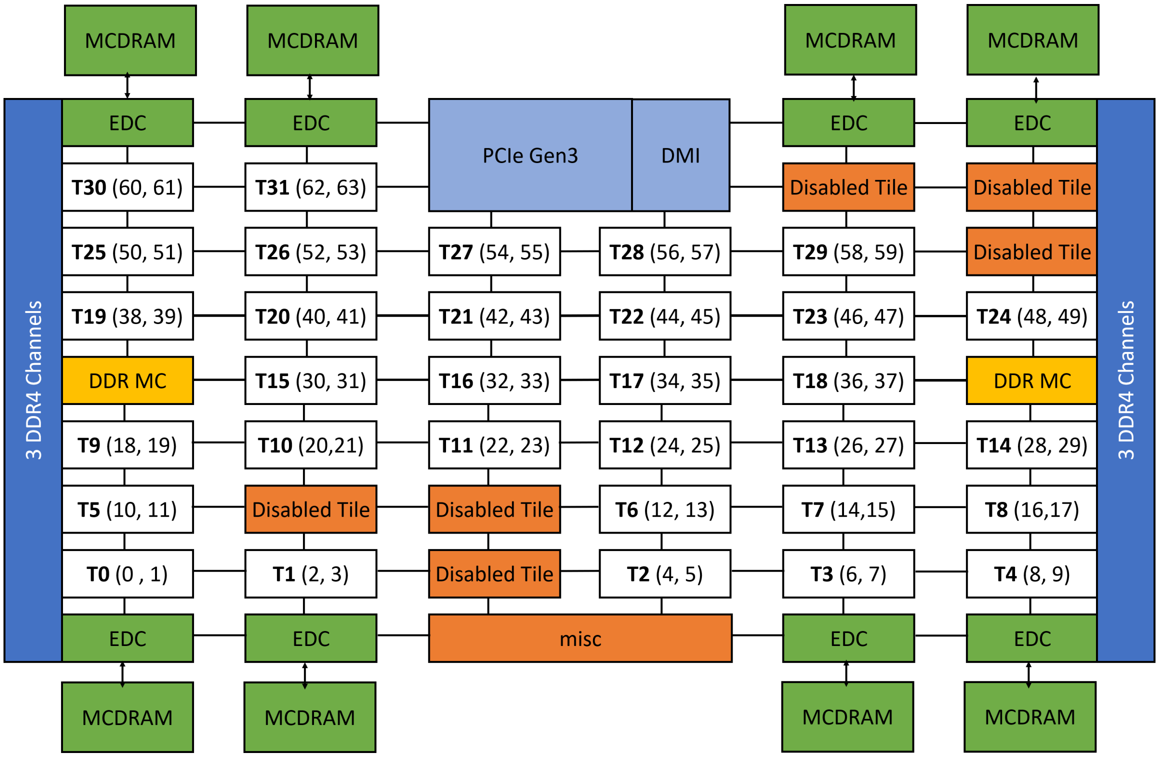


Figure 5.2: Intel Xeon Phi 7210 Knights Landings with 32 tiles and 64 cores.

unique architecture and memory design. Figure 5.1 also shows how Xeon Phi can train networks faster simply by using more threads during runtime. In this work by providing an exploratory study on Xeon Phi using a deep learning framework used to run a different networks on ImageNet data. With the help of Xeon Phi's memory modes and quadrant modes, it is possible to run these networks on different combination of memory-cluster modes. Later sections provide more details on how this exploratory study is performed and conclusions drawn.

5.3 Intel Xeon Phi Architecture

Xeon Phi architecture as shown in Figure 5.2 is made up 64 low power Silvermont micro-architecture with changes catering to high performance workloads like deep learning CNNs. Each of these 64 cores is paired to form 32 tiles, with each tile or 2 cores sharing

L2 cache and also having multiple dedicated Vector Processing Units (VPUs). Network on chip routing followed by Xeon Phi is a 2D mesh YX routing. Figure 5.3 shows single tile in a Xeon Phi holding two cores, with shared 1 MB L2 cache and one Caching/Home Agent (CHA). Each core is capable of running 4 threads simultaneously using out of order cores and large translation lookaside buffer (TLBs). The 2 VPUs per core allows AVX512 instruction execution with help of fused add units and wider 512-Bit registers. Specific software calls to these VPU allows single instruction multiple data processes.

The architecture design also shows set of two types of memory channels that enable connections to DDR4 memory and another set of channels leading to Multi-Channel DRAM (MCDRAM) which is capable of providing 4x more bandwidth compared to DDR4. DDR4 can be connected using any of the six channels on the right and the left side of the architecture as shown in Figure 5.2. On other hands, MCDRAM is spread across 8 different sides with each section providing a memory of 2 GB leading to 16 GB of high speed memory to 64 cores. Each of the 2 GB MCDRAM has a dedicated embedded DRAM memory controller (EDC). The critical reasoning of such placement of MCDRAM is discussed in the next section. Xeon Phi's two main features are memory and cluster modes. These two features determine how many levels of memory hierarchy Xeon Phi can have apart from features like where the application can fetch the data from and how the network traffic will flow from cores to directories to memories to back to the core. Memory and cluster modes are discussed in Section 5.3.2 and Section 5.3.3 respectively.

Figure 5.2 also shows specific tiles that are disabled. These tiles are not pre-disabled and this information is obtained based on the physical core IDs that get logged while performing deep profiling of system architecture using kernel tools as discussed in Section 5.5. The specific version of Xeon Phi Knights Landings used for this study has 64 cores or 32 tiles. This is 8 cores or 4 tiles less than the paper [65] introducing KNL. The same paper shows a total of 36 tiles but in reality, there are 38 slides as per the block diagram of the

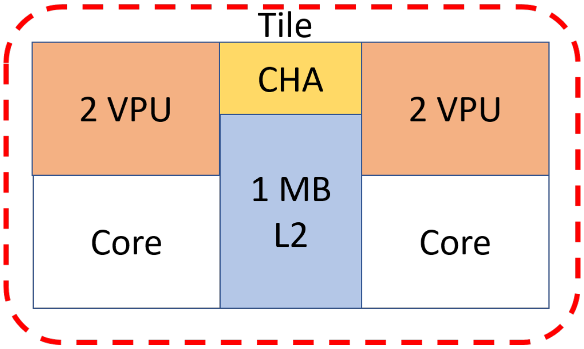


Figure 5.3: Every single tile in Intel Xeon Phi 7210 Knights Landings has two CPU cores. Each core has 2 dedicated VPU. Two cores have a shared L2 cache of 1 MB

architecture in the paper. This means there are a set of 2 tiles disabled in the Xeon Phi version used in the [65]. If 6 more tiles are disabled from same version of Xeon explained in the paper [65], one gets Xeon Phi model used for exploratory analysis in this work which has 32 tiles and 64 cores.

5.3.1 Heterogeneous Tile

Xeon Phi used in this work has 32 tiles as shown in Figure 5.3. Each of these tiles has 6 processing units of two different types. This essentially makes each of the 32 tiles in Xeon Phi nothing but a heterogeneous tile architecture. Two of the processing units shown in Figure 5.3 are CPUs running x86 and AVX512 instruction set. Each of these CPUs shares two vector processing units (VPUs). Both the VPUs and CPUs have common 1 MB of L2 cache and communicate with the help of Caching/Home Agent (CHA). With many core architecture, thermal runaway also becomes an important challenge to handle [7]. Figure 5.2 shows there are 32 tiles with each holding 6 processing units including CPUs and VPUs. To ensure that the temperature is not a critical issue with Xeon Phi, CPUs in tile are designed based on low power Silvermont Atom processor. This processor are mostly used for mobile devices but are specifically re-designed to make use of them in

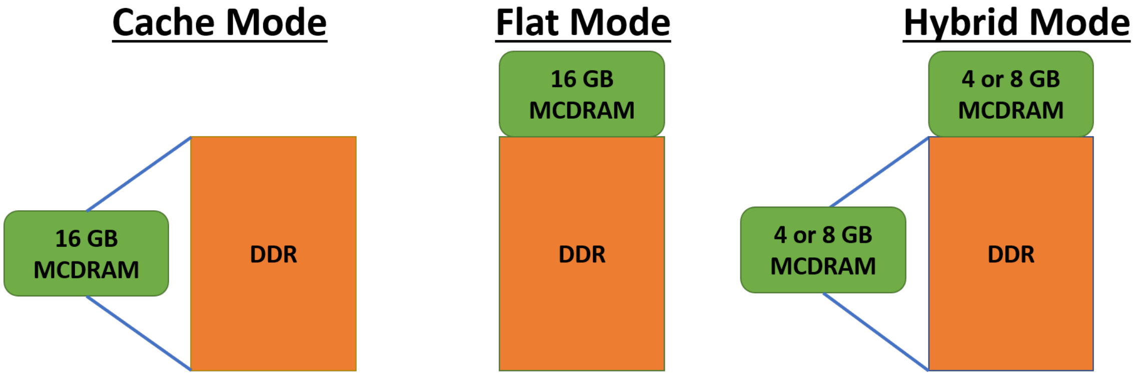


Figure 5.4: Xeon Phi 7210 Knights Landings during boot time can be configured to run the memory in three different modes - Cache, Flat, and Hybrid

large number in Xeon Phi.

The VPU shown in Figure 5.3 uses a novel 512-bit SIMD instruction set, which is specifically designed for many core x86 architecture like Xeon Phi. It also has modified multiply-add instructions that allow execution of multiple floating point operations per cycle. Many of the mathematical operations can be offloaded to VPU as it has inbuild extended math unit to perform calculations on fly that traditional CPU units would take multiple cycles leading to performance degradation.

5.3.2 Memory Modes

Xeon Phi is equipped with a high bandwidth memory called MCDRAM and regular DDR4 memory. MCDRAM is capable of providing up to 400+ GB/sec of speed and storage size of 16GB. DDR4 in Xeon Phi can provide up to 90+ GB/sec of transfer speed with a storage capacity of up to 300 GB. The system used for exploratory analysis has 100 GB of DDR4. In total, the Xeon Phi's 64 cores are exposed to 100 GB of low level memory and the 16 GB of memory provided by MCDRAM can be used in three different modes, also known as memory modes. Figure 5.4 shows three types of memory modes that can be used to make MCDRAM act either as L3 cache or last level memory. The three different memory modes in Xeon Phi are - *Cache Mode, Flat Mode, and Hybrid Mode*.

Cache Mode: When Xeon Phi is used with cache mode all of 16 GB of MCDRAM acts as L3 cache. This is an important feature for any application that is going to work with a maximum of 16GB of data as it can directly bring all the data to L3 and any L2 cache miss will lead to reducing cycle penalty. This is due to the data being present in L3 cache which is the next reference point. While using Xeon Phi in cache mode it is important to map the data to MCDRAM using either memory allocation functions or pre-built NUMA control tools. Figure 5.4 illustrates cache mode. It shows how MCDRAM acts as 16 GB of cache. Memory allocation of all application running during cache mode starts by default at DDR4. To make use of cache mode feature, applications should start memory allocation from MCDRAM. Otherwise running in cache mode will not provide a performance improvement.

Flat Mode: Xeon Phi in flat memory mode makes use of all of 16 GB of MCDRAM as last level memory and the memory address allocation for workloads by default starts at MCDRAM and then interleaves to DDR4 if an application requires more than 16 GB of memory. Figure 5.4 shows a flat mode structure. Since the target system has 100 GB of DDR4, the total last level memory available for application is 116 GB due to the addition of MCDRAM. The flat mode can provide better performance if used with correct cluster mode other. Based on the results discussed in Section 5.6 flat mode provides better results when used with any other cluster modes than All-to-All.

Hybrid Mode: In hybrid memory mode both cache and flat memory are combined to provide virtual cache and flat memory. When set to hybrid mode during boot, the system will see half of the MCDRAM being used as L3 cache and another half as last level memory. Hybrid mode becomes useful if used with sub-NUMA SNC-2 and SNC-4 clustering mode due to a direct affinity between core to a directory to memory. The target system used in this work has a total of 16 GB of MCDRAM. During hybrid mode, half of the MCDRAM

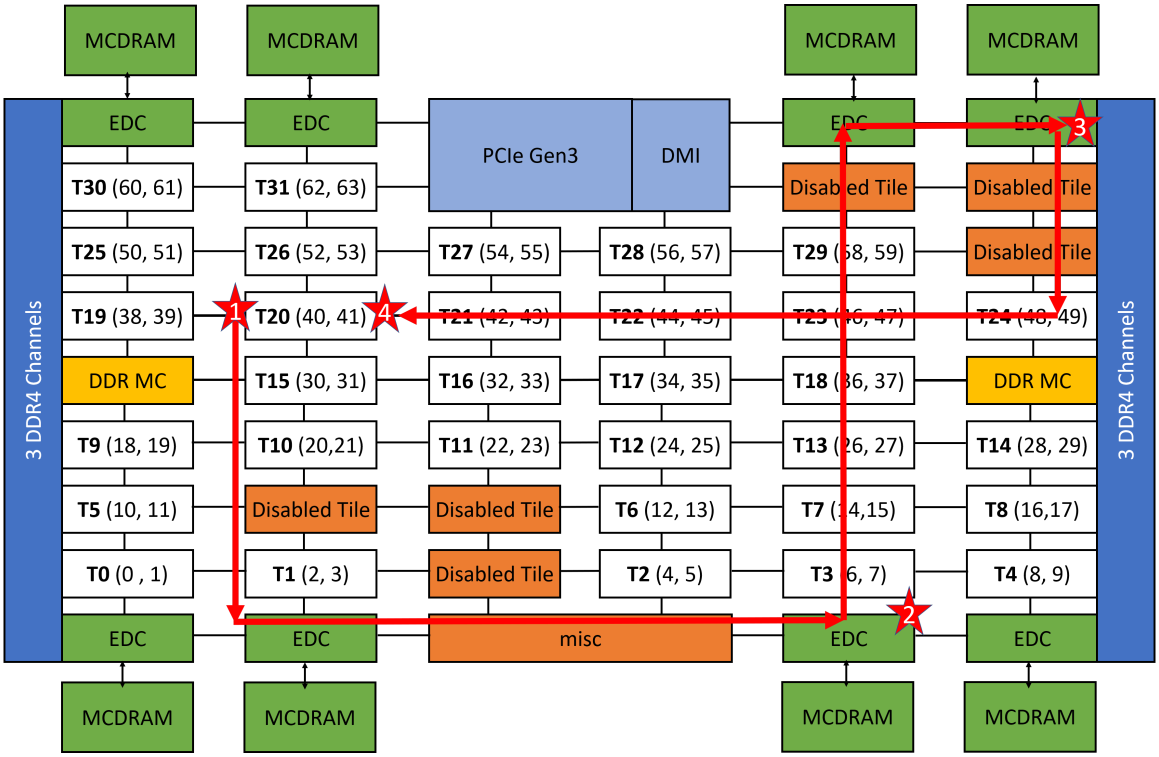


Figure 5.5: Xeon Phi 7210 Knights Landings working in All-to-All cluster mode. Memory is shared across all tiles/cores. No virtual quadrant exists in All-to-All.

(8 GB) is used as cache and other half acts as last level memory.

All three memory modes are capable of providing a performance benefit. However, performance gain also depends on how big the memory footprint of the workload is going to be and more importantly what cluster modes, as discussed in Section 5.3.3, Xeon Phi is operating on. A correct combination of memory with cluster mode can lead to more controlled traffic movement and enable faster execution.

5.3.3 Cluster Modes

Apart from the ability to provide different types of memory modes, Xeon Phi can also be configured to run in different cluster modes. The major difference between memory and cluster modes is that when a specific memory mode is used then Xeon Phi provides

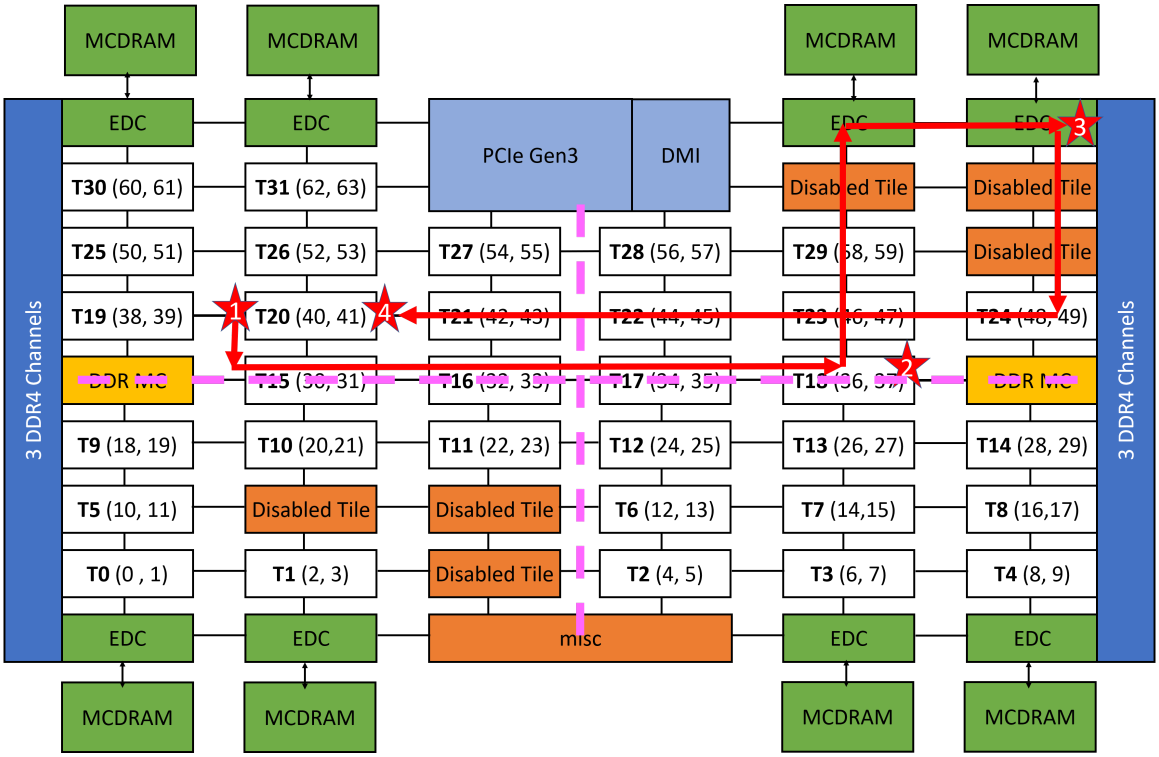


Figure 5.6: Xeon Phi 7210 Knights Landings packet movement when used in Quadrant cluster mode. There are four virtual quadrants with each having one fourth of the total MCDRAM and DDR4 memory.

application ability to make use of the memory architecture for address allocation and data management. However, based on which cluster mode is active Xeon Phi can route the traffic from core to memory differently. Based on the affinity between tile, directory, and memory, clusters can provide varying performance. These cluster modes can lower latency, improve data bandwidth and eventually decrease the distance travelled by the packets within a chip architecture. These modes are adjustable from the basic I/O system (BIOS) at boot time similar to memory modes. Xeon Phi can be used in three different cluster modes - All-to-All, Quadrant and Sub-NUMA clustering (SNC-2/SNC-4). This section provides in depth detail about these three types of cluster modes and explains how these mode can help gain performance.

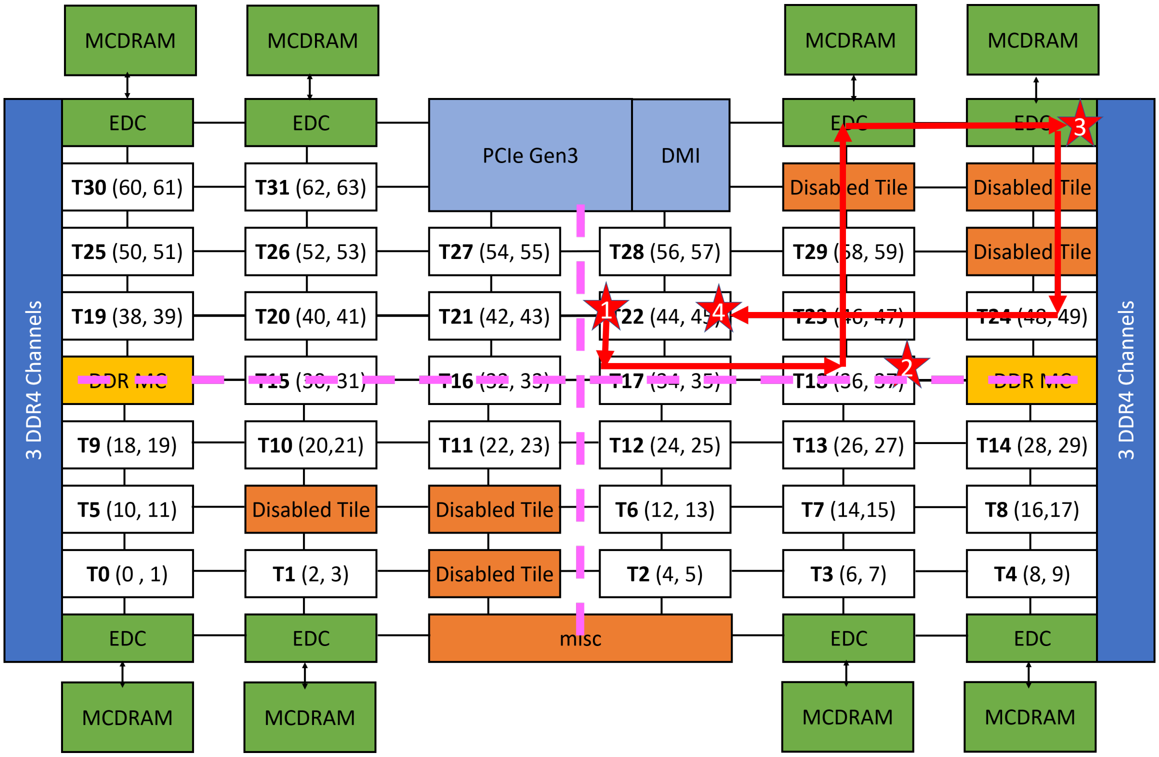


Figure 5.7: Xeon Phi 7210 Knights Landings packet transfer during SNC cluster mode. SNC has two sub-modes, SNC-2 and SNC-4. SNC-4 is similar to Quadrant mode. SNC-2 sub-mode has two virtual quadrants instead of four and has a direct tile to the core to memory affinity.

All-to-All Cluster Mode: This is the default cluster configuration used by Xeon Phi. All-to-All mode has a fundamental drawback as it does not have any affinity among tile, directory, and memory. As shown in Figure 5.5, All-to-All works on the principle that a miss hit occurring in any tile does not mean that the nearest memory will hold the data requested. This typically leads to lower performance compared to other clustering modes. As Xeon Phi is essentially a mesh network that follows YX routing, a miss on tile T20 (T stands for tile) in Figure 5.5 means that the request packet has to first traverse Y direction from (1). In this example, the directory affinity to T20 is the EDC marked (2). After traversing in Y direction from tile T20 the packet then traverses to (2) in X direction where it gets the memory address for the data request by tile T20, which is (3). From (2) the packet then follows Y direction as per YX routing until it reaches another EDC node and then moves in the X direction and gets the data requests by T20. From there by again following the YX routing the packet eventually reaches back to (4) which is the requesting tile T20.

This clearly showcases that when Xeon Phi is fully loaded with multiple data request the mesh network will get busy. This will eventually impact performance. There are only two ways to ensure this does not occur. First is by mapping threads and data such that it is closest to the tile running the thread. This also means making use of other cluster modes that provide better affinity. Second is by running less memory intensive workloads, which means running workloads that are compute intensive rather than memory intensive.

Quadrant Cluster Mode: When BIOS is enabled to run Xeon Phi in Quadrant cluster mode then the architecture is divided into four virtual quadrants as shown in Figure 5.6. When miss hit occurs at tile T20 (marked as (1)) in Quadrant mode with EDC (marked as (3)) holding the data, the traverse path is very similar to All-to-All. However, in this cluster mode, there is a direct affinity between directory and memory. This also means that a request from a tile can go to any directory, but the directory will only access memory from the same quadrant. All the four quadrant are allocated an equal amount of last level memory to ensure symmetry. Quadrant mode performs better compared to All-to-All as the latency of request is less and memory regions are visible to the workloads that can ensure correct thread to the quadrant to memory mapping. For the system used in this work, the capacity of DDR4 memory is 100 GB and MCDRAM is 16 GB. As soon as BIOS is enabled to run Quadrant mode the memory is equally divided in to four equal memory with each quadrant allocated 4 GB of MCDRAM and 25 GB of DDR4 dedicated to it.

There are clear benefits of using Quadrant mode over All-to-All, however it also depends on the correct mapping of threads which is crucial in ensuring that the L2 miss request is catered faster to provide better performance. By default a thread running in a quadrant, bottom left section in Figure 5.6, will get memory allocated in two of the MC-

DRAM located with in the same region. If memory mode is either cache or hybrid then dedicated L3 cache is provided to a specific number of cores/tiles within the quadrant.

Sub-NUMA Cluster (SNC) Mode: The last of the three cluster modes is Sub-NUMA (non-uniform memory access) Cluster (SNC). SNC mode within itself has two sub-modes - SNC-2 and SNC-4. In this mode, each tile has a direct affinity to directory and memory. This means a request originating in top right section/quadrant, as shown in Figure 5.7, will ensure that the data resides in the same quadrant which is a major improvement over Quadrant. By taking an example of a request originating from T22 marked as (1) in Figure 5.7 will mean that the MCDRAM within the quadrant will hold the data and this information is also stored in the directory within the same quadrant. When SNC-2 cluster mode is enabled then same properties apply to two quadrants which are bottom half and the top half in the same figure. In SNC-4 the tiles are virtually divided into four different quadrants as clearly marked in Figure 5.7. The amount of memory shared in SNC clustering mode is exactly similar to Quadrant mode with the difference that when in SNC-2 each of the two quadrant gets 8 GB of MCDRAM and 50 GB of DDR4.

SNC cluster mode is the most advanced of the three modes. It has dedicated affinity between tile, directory, and memory, leading to a faster performance by reducing L2 miss penalty. Due to the software ability to map threads to cores and accordingly allocate data to memory within the same quadrant, SNC is supposed to provide far better performance compared to other two cluster modes.

5.3.4 Thread Modes

Mesh routing, cluster mode, and memory modes are unique architecture features from the hardware point of view which Xeon Phi is capable of providing. However, since Xeon Phi has 64 cores available to the application, it is also important to understand how the workload running on Xeon Phi can be mapped by making use of thread affinity and thread

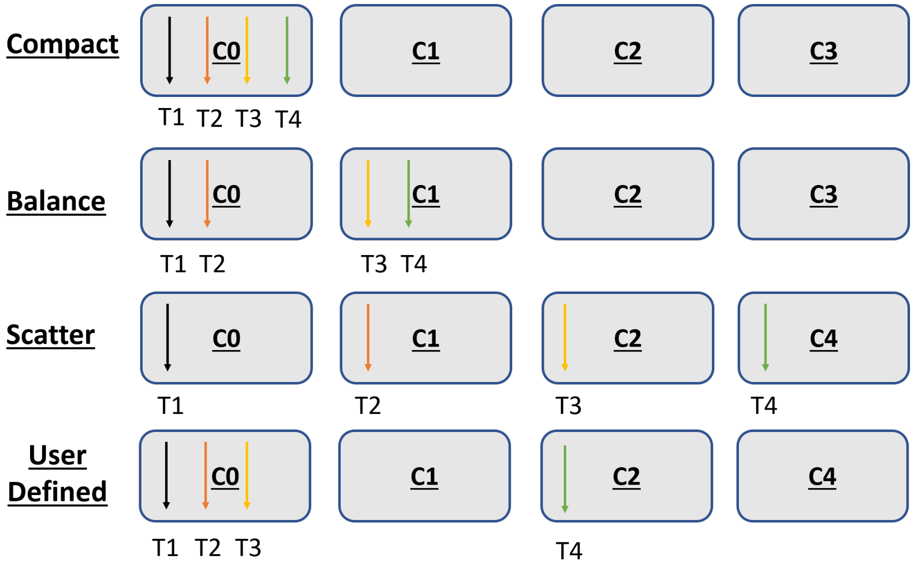


Figure 5.8: Xeon Phi 7210 Knights Landings allows threads to be distributed and affinity to be set in four different manners. It gives flexibility to an application during runtime by allocating resources based on the number of threads to run.

distribution. Using system level libraries it is possible to run the application written or compiled with OpenMP calls to manage threads in four different possible ways - Compact, Balance, Scatter, and User Defined. In this section, thread affinity is discussed which controls, where the thread will run and then thread distribution is explained which controls how threads are distributed based on the number of threads to execute.

Thread Affinity: If an application has more than one thread then it is possible to map these threads such that it can reduce run time. When running such application on Xeon Phi, depending on the number of threads each thread can be tied to a tile and within the tile which core will run that thread can also be decided. This is achieved by running an application with the help of OpenMP. The application running should have a code structure that allows OpenMP thread management. Consider an example of four threads in an application running on Xeon Phi as shown in Figure 5.8. It is possible to run all the four threads on Core C0 and it is also possible to run the same four threads on core C0 and core C1 with two threads per core. Such management of threads is irrespective of which type of thread distribution is enabled and also thread affinity is in some sense a manual process as the threads can be asked to run on specific core thread ID. This process can be automated by making use of thread distribution.

Thread Distribution: For Xeon Phi, threads can be managed by making use of four different types of distribution techniques without worrying about manually mapping threads to cores. It is achieved with application and tools written with OpenMP APIs. Figure 5.8 clearly shows the difference between the four types of distribution techniques. For simplicity, consider that there are four threads to run and the same number of cores are available. If the application is enabled to run in Compact thread mode then all four threads will be sequentially allocated to the core C0 only since each of the four core can run a maximum of four threads. This also means thread allocation starts at first core available and moves to next core only if the core already has a maximum of four threads running. If the same four threads are to run in Balance mode then total number of threads are divided into half and are distributed to core C0 and C1. This makes core utilization better compared to Compact. During Scatter thread mode the number of threads will be divide by four (or the number of physical cores on architecture) and then allocated to one core at a time sequentially till all threads are mapped to at least one core. With a dedicated core for each thread, computation is much faster. The last thread distribution criteria are User Defined which can be mapped as an example in Figure 5.8. This means one can provide more or less thread to one core and on another hand, another core may be running half of the thread the first core is running. This also means the application can dictate how and where the threads will run. However, for this, to happened the application has to be hardcoded with thread to core mapping.

By introducing thread management with cluster and memory mode Xeon Phi provides numerous ways using which performance of any workload can be improved. By making use of three different memory modes with four cluster modes to run threads in four different ways runtime can be reduced drastically. In later sections, deep learning framework, networks and data used to analyze these different types of architectural modes is discussed.

5.4 Deep Learning Framework, Networks and Data

The exploratory analysis carried out as part of this research work is achieved by running different types of CNN networks [40] [28] [63] [67] on a deep learning framework [33] using ImageNet [16] open data set. This section provides a brief overview of which tools were used and the reason to use those. The amount of data collected has enabled deep insight into Xeon Phi's architectural features.

5.4.1 Experimental Framework

Caffe [33] is one of the most widely used deep learning frameworks. It has numerous features that make it easy to use across different types of deep CNN networks. Networks running on Caffe can be used to train and validate models that use images as input. These images can be from any open dataset and one such dataset used in this work is ImageNet [16]. For Xeon Phi, Caffe has already been optimized by Intel. It is done by inserting API calls within Caffe that allow thread distribution using OpenMP thread management as discussed in Section 5.3.4. The benefit of such optimization is that the framework can be used on Xeon Phi with thread optimization by running N number of threads across different networks. Another added advantage of such optimization is that the neural network being used requires no modification. This ensures that the network being used for architectural profiling is the original networks with original training parameters. Optimized Caffe itself has shown great performance benefits. However, so far none of the studies has explored the effect of the clustering, memory and thread mode in terms of reducing time to train network and that too on vast amount of data like ImageNet ILVSRC12 [60].

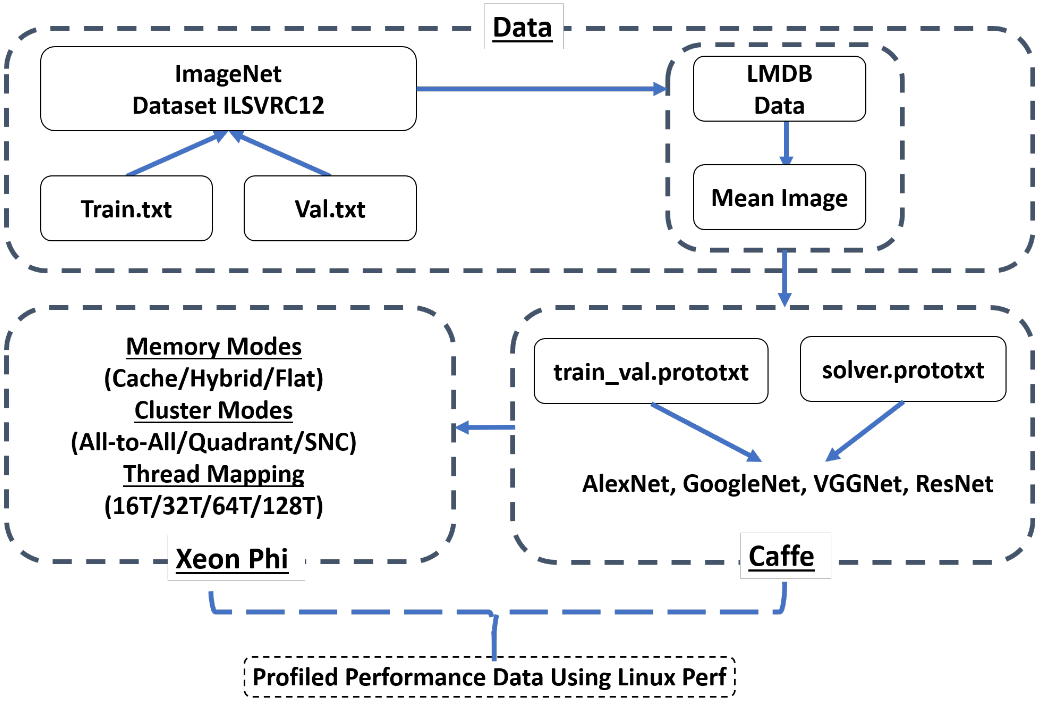


Figure 5.9: Experiment setup showcasing how ImageNet data is used in Caffe framework to train different networks using various Xeon Phi tuning parameters such as memory, cluster and thread mapping.

This framework also provides easy setup of data that can be used to train the network. The data preparation details and pre-built scripts can be directly applied to the raw data. Once the data is available in the format as per Caffe requirement, then different networks can be evaluated without modifying anything on the framework side. Such fast and easy to use framework with software architecture providing options to run threads in Scatter, Compact and Balance mode allowed detailed training and evaluation of the architecture.

5.4.2 Neural Networks Under Study

Neural networks are used to train image based data to come up with a trained model that can be applied to unseen data. This is the de-facto process used in the field of artificial intelligence and machine learning. These networks are written in a specific format and to use them one has to use deep learning frameworks like Caffe. There are numerous networks proposed over the last decade, but only a few of these have attracted other researchers to come up with networks to better the accuracy as achieved by previously proposed CNN networks.

In this work, four specific networks are used - ResNet [28],AlexNet [40],VGGNet [63] and GoogleNet [67]. For the sake of simplicity, the networks used in this work are not modified. All learning parameters are kept as per the original network settings. This is to ensure that there is no run to run variation. The major reason to use these four networks is that each of these provided elegant solution during different years of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [16]. This made it interesting to see if the same networks when running on an architecture like Xeon Phi can take less time to train by tuning system parameters like memory modes, cluster modes, and threads. On Xeon Phi, these networks are used as the default network structure is and only parameters that were changed in these networks were the number of threads. Each of these four networks makes use of the same data set with the same number and sequence of images to make the experiments more consistent.

5.4.3 Data Collection

To evaluate Xeon Phi using deep learning framework, structured data is required. Due to the ImageNet challenge the vast amount of open data set is readily available for anyone to download and use without any restrictions. For the exploratory analysis in this work ImageNet Large Scale Visual Recognition Challenge (ILSVRC) data set is used [60]. The data is available in raw format and separated as training, test and validation data. Caffe provides process to convert these raw files into Lightning Memory Mapped Database (LMDB) format. This ensures there are no bottlenecks due to data fetching during training phase. With more than 10 million images covering 100 different object classes ensures that the network being developed or used is accurately trained. Such a huge amount of data also allowed Xeon Phi memory to be fully loaded which also increases traffic network under different cluster modes.

5.5 Instrumentation And Experimental Results

To correctly explore Xeon Phi architecture, an experiment setup that makes use of Caffe, networks and ImageNet data is created. The major goal is to enable an experiment environment that provides profiling data. To achieve this, Linux tools were used. The platform used for experimentation is DAP [57] that came with Intel Xeon Phi 7210 processor with 32 tiles having 2 core each, providing a total of 64 cores. Each core can clock a maximum frequency of 1.5 GHz in turbo mode. The operating system running on DAP is CentOS 7.3 configured with Xeon specific software package 1.5.3. The system is also configured with Intel deep learning math libraries. This is required to make use of AVX512 architecture instruction set. Xeon Phi is traditionally not designed to be energy efficient hence the governor policy is kept by default to run at the highest frequency of 1.5 GHz. One of the tools which were extensively instrumented for this work is Linux Perf [37]. It allowed profiling without the need to modify any of the deep learning source code. Another tool used is numactl that provides flexibility to map application to desired section of the memory. This is useful when the system is configured to run in different cluster (mainly Sub-NUMA clustering mode) and memory modes. Figure 5.9 provides a detailed overview of how the experiments on Xeon Phi were carried out in order to capture the important performance details.

Data Preparation: ImageNet data used for experimentation is a set of two groups of raw JPEG images covering different types of objects. The two groups in which the images are provided are training and validation images. During the training of a network training images are used and during the same training process periodically validation images are

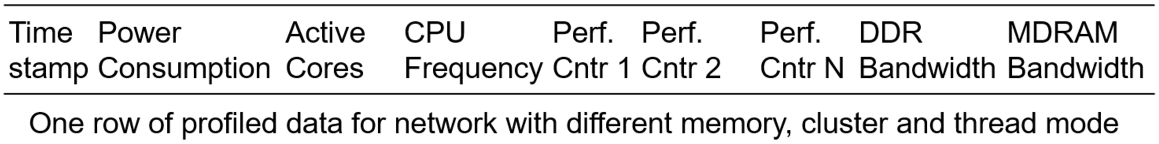
Table 5.1: System and application level performance data collected in this work.

Application Parameters	System Parameters
Instructions Retired	CPU Frequency
CPU Cycles	Number of Active Cores
EDC Controller Request	DRAM Power Consumption
DDR Controller Request	Package Power Consumption
Data Memory Access	DDR and MCDRAM Bandwidth
Execution Time	

used to validate the trained model to calculate loss and adjust the learning parameters. The images retrieved from ImageNet are first reduced to a specific size of 256x256. This is achieved by pre-built scripts in Caffe which ensures that the quality images does not suffer and the compression achieved reduces the time to fetch the data from low level memory including hard drive where the big data is stored. By reducing image size more images can be stored in a cache (when MCDRAM is used in cache mode) and it can reduce time to train by fetching new images faster. The data section in Figure 5.9 shows exactly how this process is achieved. After reducing the size of images, all the images are combined and stored in a Lightning Memory Mapped Database (LMDB) format. LMDB stores the image sequentially in the manner they were labeled using database format. This allows faster access to images. The last step of data preparation is the computation of the mean of all the images. All the training model requires subtraction of image mean from each image which is used as a training parameter.

Memory and Cluster Combination: Similar to the different number of threads, the experiments also made use of all available types of memory and cluster modes. Using BIOS settings, for every possible combination of memory and cluster mode network is trained on all combination. In all, there are nine combinations of memory cluster mode, but in this

Table 5.2: Data format for each time data is logged using Linux Perf.



Time stamp	Power Consumption	Active Cores	CPU Frequency	Perf. Cntr 1	Perf. Cntr 2	Perf. Cntr N	DDR Bandwidth	MDRAM Bandwidth

One row of profiled data for network with different memory, cluster and thread mode

work, SNC-2 is also used as one of the cluster modes that increased the total number of experiment combination to twelve.

Thread Distribution: All the networks used for experiments were run with 16T, 32T, 64T, and 128T. For each run data is collected with four different thread settings. This provided data that gave insight by loading the network of Xeon Phi with increasing traffic. In the experiment and results section T in 16, 32, 64, and 128 stands for threads. Exploratory analysis is carried out with the combination of different types of thread distribution technique. This meant experiments for 16T were also ran first with Scatter, then Balance and finally Compact thread setting. Such a methodology allows that the exploratory analysis covers all aspect of how Xeon Phi can be used by applications.

Setting Up Caffe To Train Networks: To use Caffe correctly first important step is data preparation and second it to configure networks correctly. All the four networks - AlexNet, GoogleNet, RestNet, VGGNet - used in this work were set with default training parameters. The only parameter that is tuned is the batch size. For all the networks batch size is kept constant at 256 to avoid run to run variation with respect to data. Training and validation images were pointed to LMDB data files to ensure that the same data is used by different networks during the training period. Compilation of Caffe by done using Intel deep math libraries and more importantly with compiler and libraries that allowed thread distribution. Total number of iteration for all the networks is set to 100. This is done to ensure that the training is completed within a reasonable amount of time without compromising the amount of traffic that each run with generating. For each iteration with a batch size of 256,

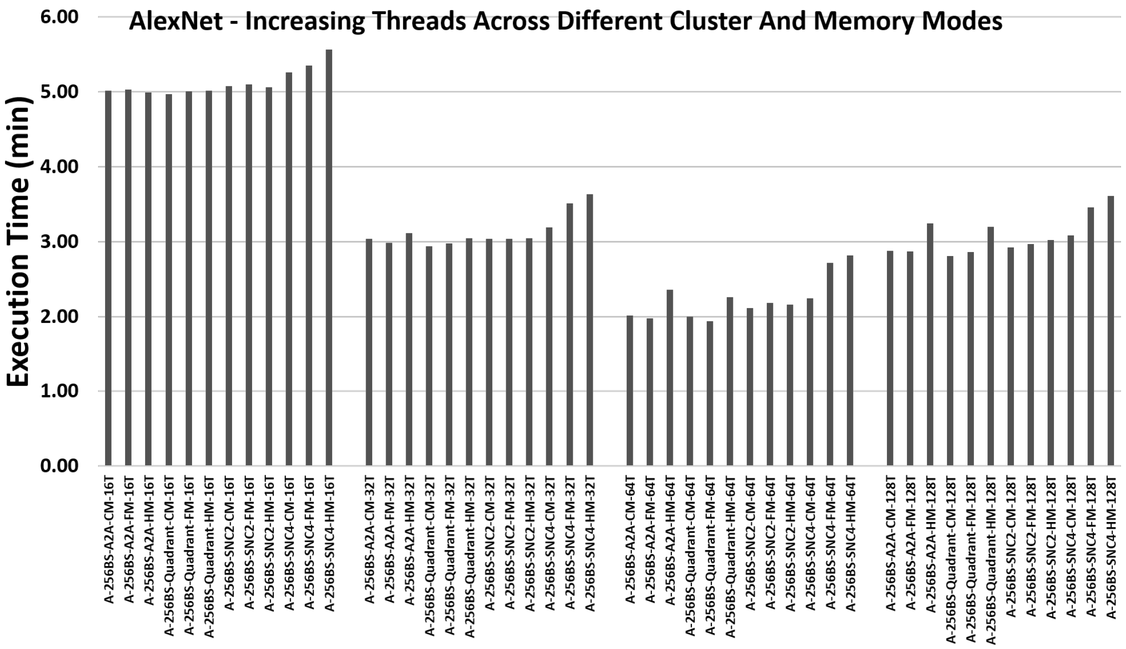


Figure 5.10: Execution time of AlexNet with 100 iteration and 256 batch size on different memory and cluster mode with increasing number of threads.

256 images were fetched.

Profiling Networks On Xeon Phi During Runtime: Final step is to make use of the profiling tool that can accurately profile network running on the system. This is achieved with the help of Linux Perf [37]. To enable more architecture specific data collection and profiling, Linux Perf is modified to read performance counters that were not by default enabled in Linux Perf tool for Xeon Phi. The profiling is set to time based where data is logged into a text file every one second. The granularity is kept as one second due to the number of counters getting logged and also since the workloads are deep neural networks it takes a lot of time and reducing profiling granularity would mean a lot of noisy data. The overhead added with the profiling is negligible as the performance monitoring unit (PMU) in x86 architectures including Xeon Phi is capable of providing multiple counters at the same time without adding overhead.

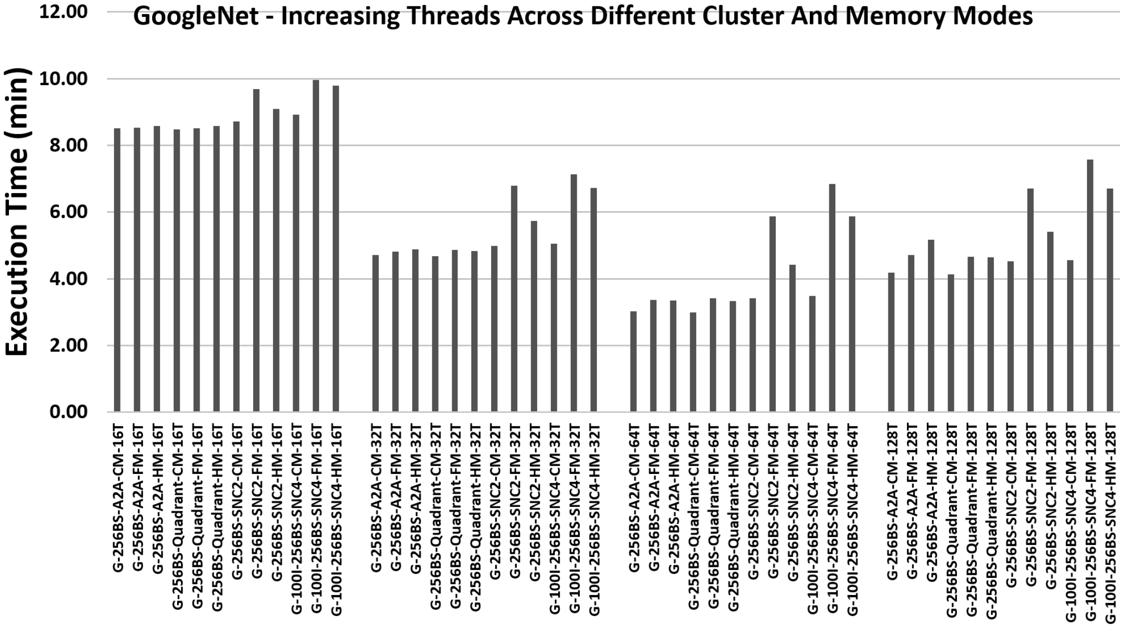


Figure 5.11: Execution time of GoogleNet with 100 iteration and 256 batch size on different memory and cluster mode with increasing number of threads.

Table 5.1 provides a list of counters that were collected every second. These are divided into application and system parameters. The major performance counters that were collected for every second of training were DRAM and Package power consumption, and DDR and MCDRM bandwidth usage. These four parameters provided how busy the system is during the training period and how many requests are being sent to a lower level of memory. Every second the data collected is logged in a text file in row format as shown in Table 5.2. Such level of data information allowed accurate analysis of how the Xeon Phi is responding to the request of the application running under different memory, cluster and thread modes. Section 5.6 presents results and analysis of all these combinations of runs on Xeon Phi.

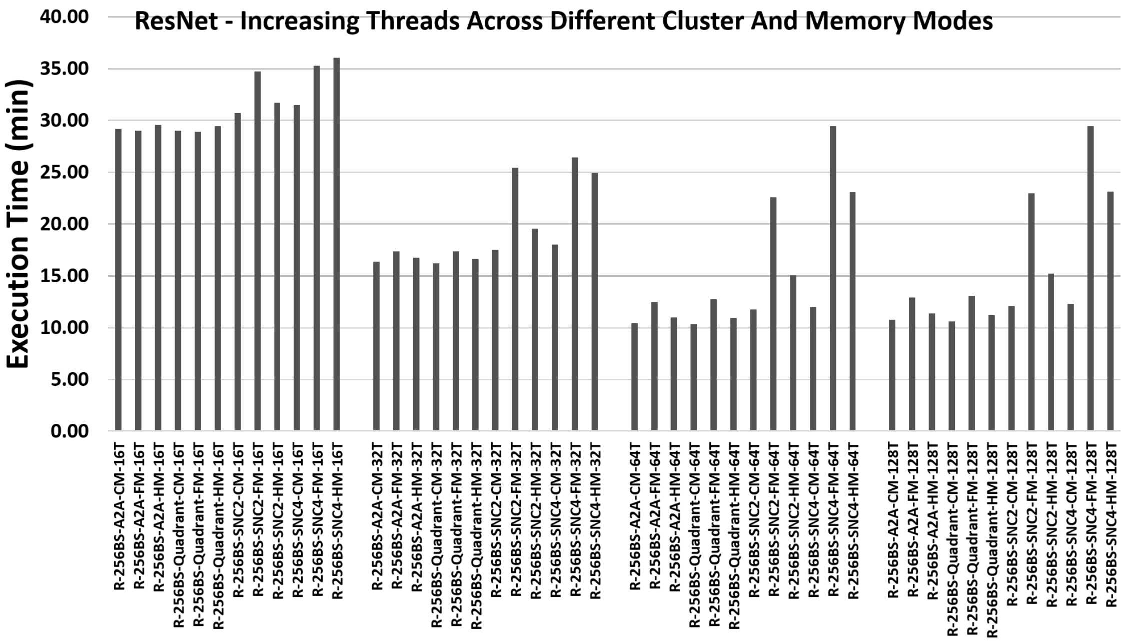


Figure 5.12: Execution time of ResNet with 100 iterations and 256 batch size on different memory and cluster mode with increasing number of threads.

5.6 Results

The experiments carried out to explore Xeon Phi architecture involved running a combination of a different memory, cluster, and thread modes. The preliminary analysis done is to understand whether the distribution of threads (Scatter/Compact/Balance) has any impact on the runtime of the application. Since the workload used for experiments are deep learning neural network trained using the large number of training images, thread distribution is not taken into account. The reason is due to the fact that a minimum number of threads that were to run is 16 and Compact distribution mode will provide lower runtime as it will run 16 threads on 4 cores. On another hand the same number of threads in Scatter mode will run on 16 different cores. Since the minimum number of threads used in this experiment Compact will perform worse than Scatter, Compact mode is not used during experiments. Balance and Scatter are no different other than neighboring threads are run-

Table 5.3: Different Xeon Phi features using which experiments were carried out.

Memory	Cluster	Number of Threads	Thread Mapping
Cache	All-to-All	16T	Scatter
Hybrid	Quadrant	32T	
Flat	SNC-2	64T	
Cache	SNC-4	128T	

ning on the same core with Balance. However, since the workload is essentially fetching and training different images every iteration Balance provides the same working as Scatter. To avoid re-running similar experiments for no performance gain, Table 5.3 shows different parameters on Xeon Phi that were used while training four different types of neural networks on Caffe.

Xeon Phi is stressed with 48 different experiment setting for the same network. Across four network 192 different types of experiments were carried out and for each second of time spent by these networks on Xeon Phi performance data is collected and analyzed. In all the results figure the X-label follows acronym that uses A for AlexNet, G for GoogleNet, V for VGGNet and R for ResNet. Threads are labelled as T, suffixed to the number of threads. For example, 16T means 16 threads. Cluster modes are labelled as A2A for All-to-All, Quadrant is for Quadrant mode, SNC-2 and SNC-4 are for Sub-NUMA Clustering modes. Memory modes are shortened to FM for Flat Mode, CM for Cache Mode and HM for Hybrid Mode. All results are for 256 batch size with 100 iterations.

Execution Time: Figure 5.10 shows how AlexNet performs across different cluster and memory modes. The major difference is seen with the increasing number of threads. When 64 threads, AlexNet provides the best performance irrespective of the cluster and memory mode. However, major performance gain is achieved only with 32 threads when compared with 16 threads. If thread counts are increased to 128 then performance degrades. Such

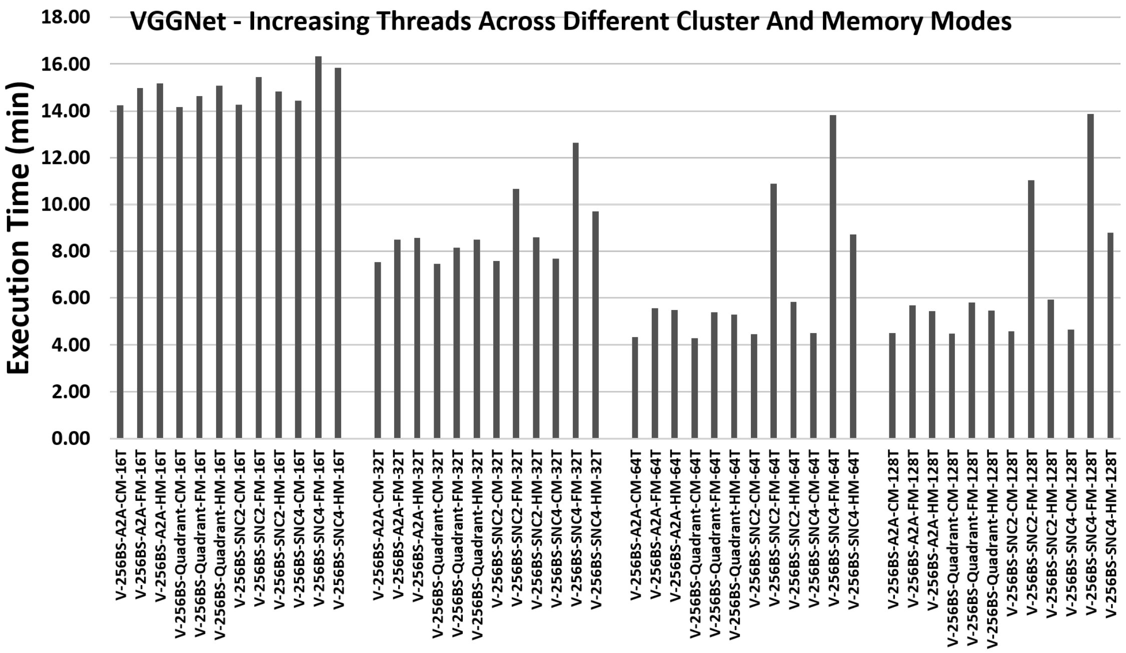


Figure 5.13: Execution time of VGGNet with 100 iterations and 256 batch size on different memory and cluster mode with increasing number of threads.

a trend is expected because each of the 64 cores is running two threads rather than one thread with 64 cores. This means cache rate decreases and improves training time. Similar trends are observed for GoogleNet in Figure 5.11, ResNet in Figure 5.12, and VGGNet in Figure 5.13.

When it comes to clustering modes, Sub-NUMA clustering (SNC-2 and SNC-4) is ideally supposed to provide far better performance due to a direct affinity between tile, directory, and memory. However, in terms of execution time, this is not the case for any of the networks as shown in Figure 5.10, 5.11, 5.12, 5.13.

All-2-all, which is the default affinity mode, is comparable to Quadrant cluster mode and provides similar performance across 16T, 32T, 64T, and 128T. In some cases, it can also be seen that SNC-2 and SNC-4 is performing worse than the other two clustering modes. The observations made for clustering mode hold irrespective of which memory mode is

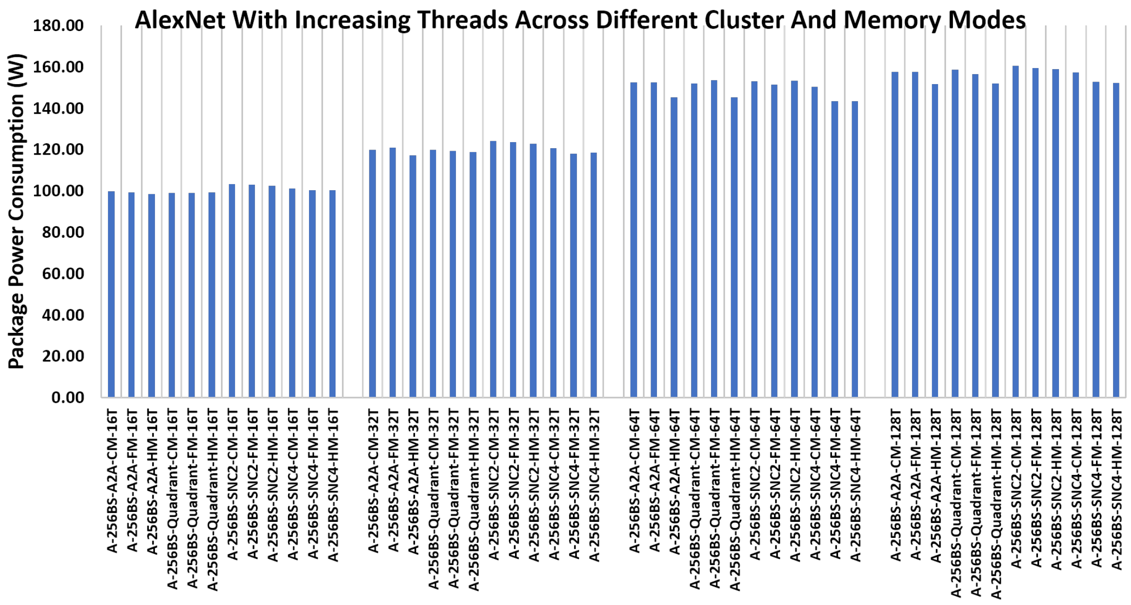


Figure 5.14: Power consumption of AlexNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

active. Cache mode and Flat mode provide better performance compared to Hybrid mode. The major reason for this is because half of the high speed bandwidth memory MCDRAM is partition into the two parts (cache and flat) which leads to address overlapping. To conclude, cluster and memory mode do provide features but that will guarantee faster training is not true. Increasing number of threads to 1 thread per core is most likely the best way to achieve better performance.

Power Consumption: Each of the network when executing on Xeon Phi is also profiled to log power consumption. Only the package power values are reported as this value is capturing power usage activity on package that has the Xeon Phi 32 tiles and high speed MCDARM memory. Figure 5.14 shows the effect of threads with different cluster and memory modes. In terms of power usage, it does not matter which clustering or memory mode is active as this parametric value is purely reliant on workload. It does not matter which network on Xeon Phi, all of them achieve the peak package consumption which is

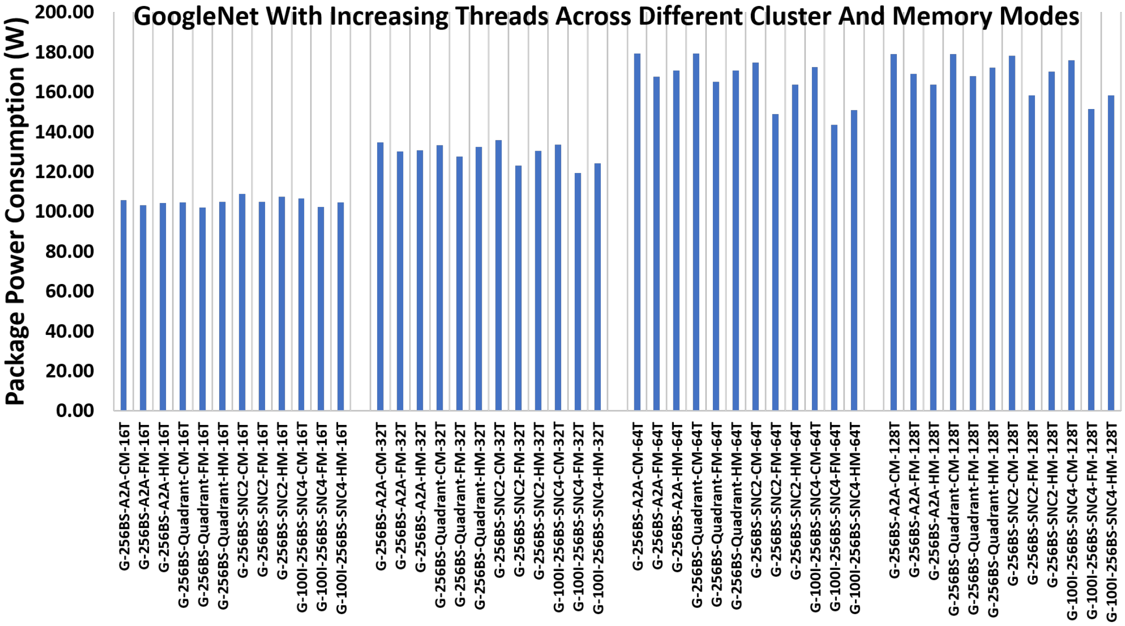


Figure 5.15: Power consumption of GoogleNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

around 180 W. Both 64T and 128T consume the same amount of power because all the 32 tiles are active.

For other networks as shown in Figure 5.15, 5.16, 5.17 similar power trend is observed. Xeon Phi is not designed to be energy efficient. The architecture is supposed to provide peak performance and it achieves that by running as many tiles it has. Energy efficiency can be achieved at the cost of performance but that is not the main goal when training deep neural networks like AlexNet, VGGNet, GoogleNet, and ResNet on Caffe. The major reason to focus on power consumption is to show that the architecture is truly busy and providing all the 64 cores to each of the threads. The trend of increasing power consumption from 16T to 128T shows that all cores are busy computing to train the network using different images.

CPI: One of the better metric to understand how a system is performing is Cycles Per Instructions (CPI). During profiling, the tool is set to log cycles and instruction retired ev-

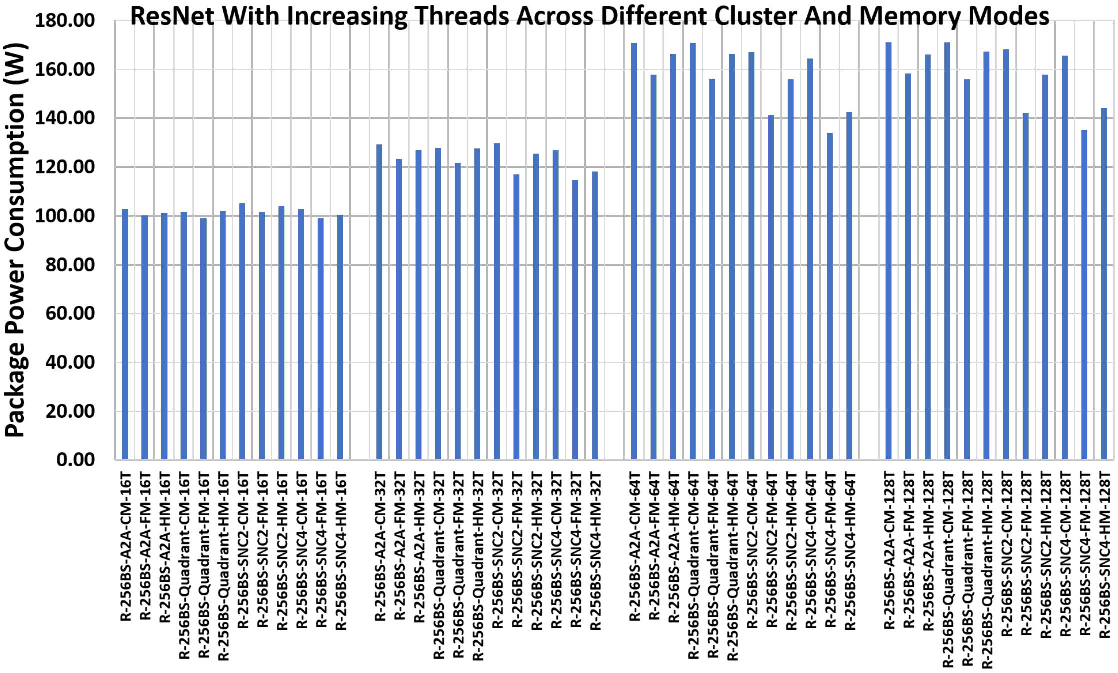


Figure 5.16: Power consumption of ResNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

ery second. Across different networks, it is observed that irrespective of which network is being trained, in SNC-2/SNC-4 clustering mode the cycles spent per instruction is almost twice. Figure 5.18 shows this trend for AlexNet irrespective of how many threads are being executed concurrently. Sub-NUMA clustering also showed a decrease in performance in terms of execution time. These two data points clearly show that Sub-NUMA is the worst clustering mode out of the three clustering modes compared to cycles spent per instructions. The major reason for such behavior is also with the fact that the data the memory is not equally distributed. In case the data being used is not spread equally across all eight controllers it might lead to slower performance as Sub-NUMA follows strict core to the directory to memory affinity. Any cross memory allocation may lead to performance degradation. Similar CPI observations are made for VGGNet, RestNet, and GoogleNet as shown in Figure 5.21, 5.20, 5.19 respectively. This also shows that it does not matter how

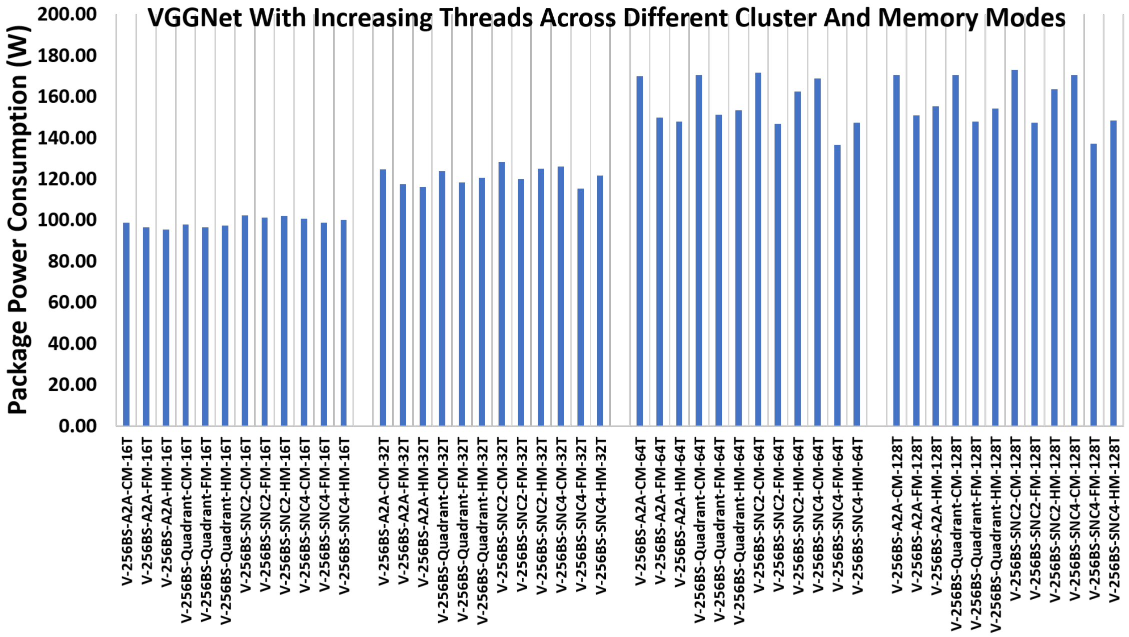


Figure 5.17: Power consumption of VGGNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

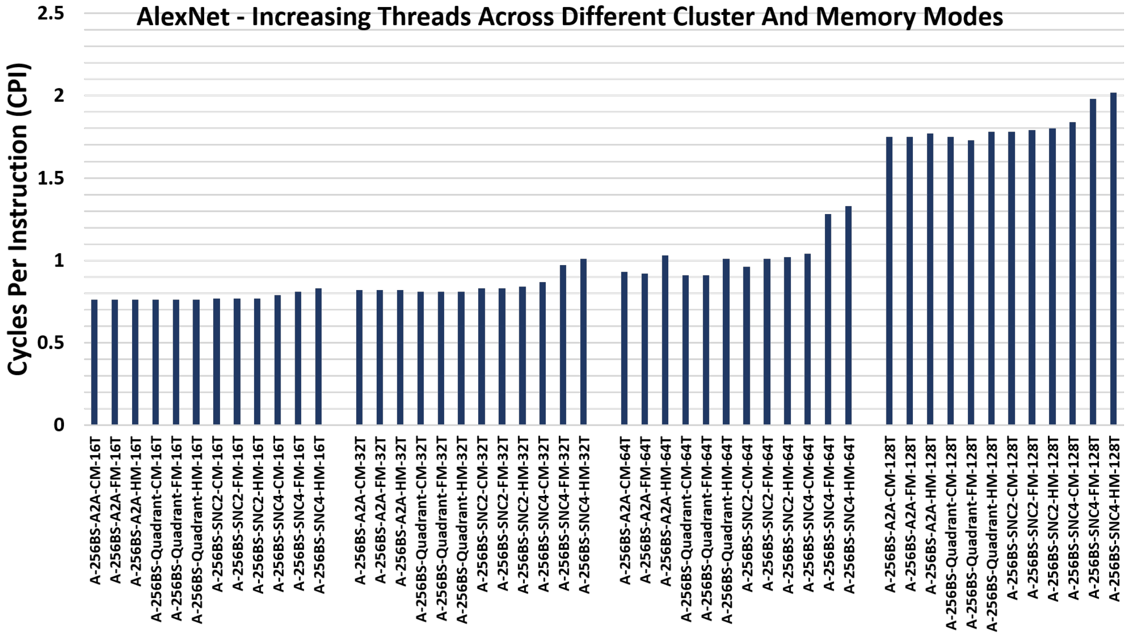


Figure 5.18: CPI of AlexNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

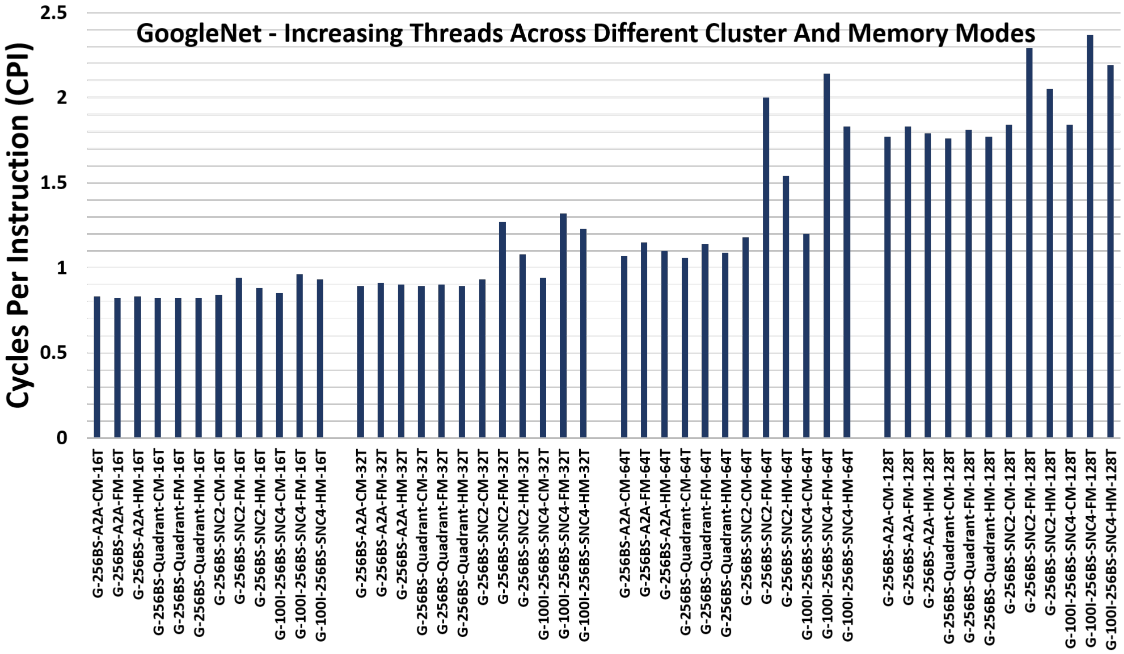


Figure 5.19: CPI of GoogleNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

big the training network is and whether it demands more activity from underlying Xeon Phi architecture, the CPI always increases for Sub-NUMA mode.

Traffic: Xeon Phi features YX routing 2D mesh networks. Figure 5.5 shows how data package travels in one of the clustering mode. The more the traffic the busy the network will directly impact the runtime of the application. For deep learning neural network training this is the major issue as many threads will be active training networks, adjusting weights, fetching data from memory, etc. Every cache miss will lead to request low level memory. MCDRAM provides more than 400 GB/sec of maximum traffic. Figure 5.22 shows how much MCDRAM and DDR4 memory bandwidth is utilized. An important observation here is that the network complexity also drives how much bandwidth utilization occurs. In the case of ResNet and VGGNet in Figure 5.22 maximum of 110 GB/sec and 100 GB/sec bandwidth utilization is achieved. This also shows that as the network complexity increases

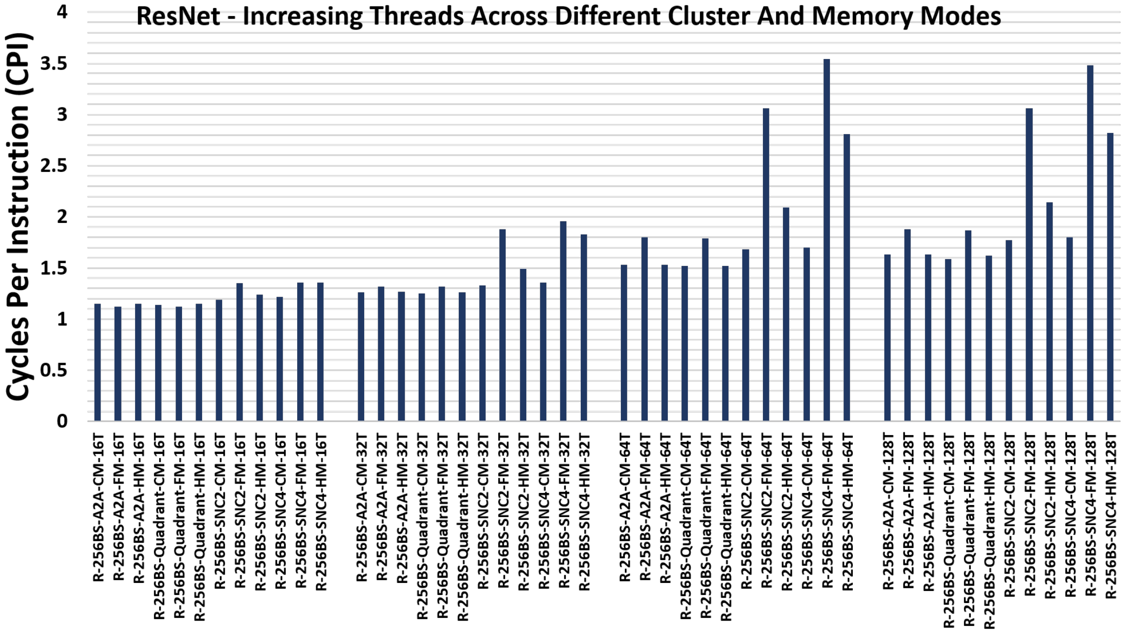


Figure 5.20: CPI of ResNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

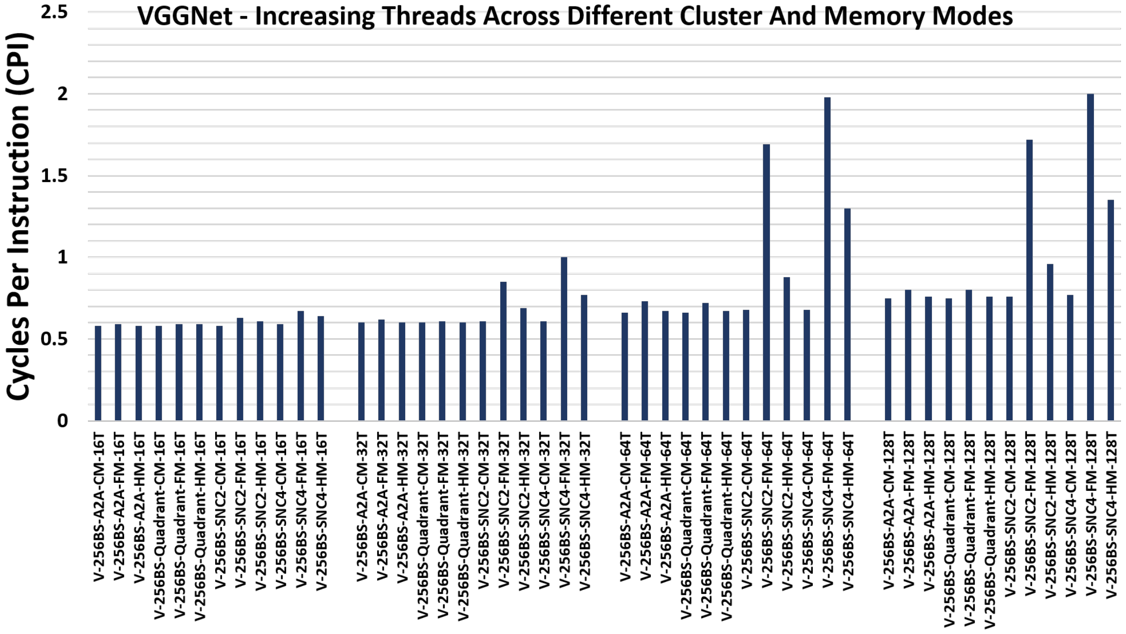


Figure 5.21: CPI of VGGNet 100 iterations and 256 batch size across different memory, cluster and thread mode.

cache miss rate increases too and it eventually leads to lower performance. At this point it does not matter which clustering or memory or how many threads are being used. Even though MCDRAM is capable of providing 400+ GB/sec of memory bandwidth it rarely achieves it. This can be the major bottleneck for an architecture with so many cores as the traffic being generated is ever increasing and the movement of these packets is affected due to the available bandwidth.

Bottlenecks: Based on the results for execution time, power consumption, CPU and memory bandwidth utilization it is clear that Xeon Phi does provide different ways to run and train deep learning networks. However, the different features does not necessarily allow it to provide better performance. The results shown conclude that the major bottleneck is most likely the tiles which are running low power Silvermont Atom cores. CPI in Figure 5.18, 5.19, 5.20, 5.21 show that the performance of cores is degrading as data traffic request increases across any combination of cluster and memory mode. Having just 1 MB of L2 caches to share among two cores and four VPUs does degrades performance. Another bottleneck is the network that is slowing the memory bandwidth usage. Figure 5.22 does show that increasing network traffic pushed memory bandwidth usage, however, it also shows that peak memory bandwidth is never touched. By making use of more powerful cores/tiles and ensuring memory bandwidth is utilize to fullest may make Xeon Phi more powerful and enable much faster training on even larger datasets.

5.7 Conclusions

The work done around Xeon Phi clearly shows the importance of detailed profiling of the system. It is important to understand any architecture not only from the software point of view but also in terms of architectural features. Section 5.3 provides a deep look into Xeon Phi architecture and different features that it comes with. Detailed exploration of memory, cluster and thread modes in Section 5.3.2, Section 5.3.3, and Section 5.3.4 respec-

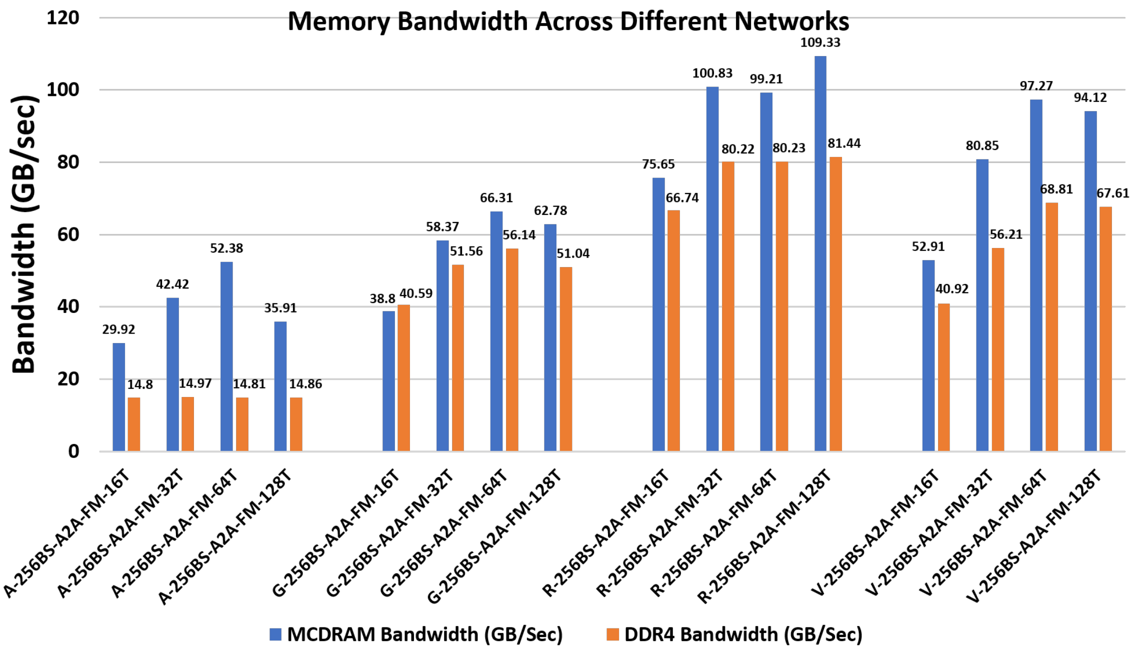


Figure 5.22: Maximum memory bandwidth across different networks with the same iteration and batch size for All-to-All and Flat Mode memory. All cluster and memory mode show a similar trend.

tively. Exploratory analysis showed how the same application can be mapped in a different manner by tuning different configuration knobs on Xeon Phi. This work also explored how the infrastructure created around Xeon Phi can be used to run a different type of neural networks and simultaneously logging critical performance details to provide deep insight in terms of performance and power. Based on the data analyzed the major bottlenecks for Xeon Phi is in the tile architecture that is running low power Atom architecture Inability to make use of maximum memory bandwidth is another reason of bottleneck in tile based architecture like Xeon Phi. With the insights drawn in this work, Xeon Phi can be further improved to provide much better performance than it currently does.

Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

Continued demand for performance led to powerful mobile platforms with heterogeneous multiprocessor system on chips. These platforms provide many voltage-frequency levels and active core configurations that can be chosen at runtime. In order to accurately propose methodologies to enable runtime optimization on heterogeneous architecture, characterized data at different level of core and frequency configuration is critical. **Chapter 3** discussed a phase-level instrumentation and characterization methodology that lead to more than 4000 phases across 19 different types of benchmarks covering widely used bench marking suites. It enabled creation of not only an instrumentation technique but also a system by which the framework can be easily used to test any offline developed polices. Two such policies were discussed in **Chapter 4**

Using phase-level offline characterization, **Section 4.2** showed a runtime optimal configuration selection policy which provided better metric, such as energy consumption. Experiments showed an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively. Similarly, **Section 4.3** presented a practical approach for dynamic management of mobile processors using the framework of imitation learning by constructing an Oracle policy to maximize PPW for a set of applications. Using this Oracle, runtime policies that are applicable to a broad range of application workloads were proposed. Experiments on a commercial mobile platform show 101% improvement in PPW on an average while running several commonly employed benchmarks. Existing governors on commercial devices employ simple heuristics based on the system utilization, which led to sub-optimal performance. As a next step, phase-level instrumentation can be extended to implement an online learning technique.

This will enable the policy to adapt itself to a wide range of previously unseen workloads.

Chapter 5 presented exploratory analysis of various features of many core heterogeneous tile based Xeon Phi architecture. Many architectural features like thread, clustering and memory modes were explored. It is shown how characterization enables deep insight into architectures with respect to deep learning. Using such technique one can easily understand the bottlenecks in different networks and also whether these bottlenecks are due to the architecture on which training is being executed or not. Such vast amount of data can also enable development of designing the architecture and making it more powerful and efficient simultaneously.

REFERENCES

[1] A. Aalsaud et al. Power–Aware Performance Adaptation of Concurrent Applications In Heterogeneous Many-Core Systems. In *Proc. Int. Symp. on Low Power Electronics and Design*, pages 368–373, 2016.

[2] R. Z. Ayoub et al. OS-level Power Minimization under Tight Performance Constraints in General Purpose Systems. In *Proc. of the Intl. Symp. on Low-power Electronics and Design*, pages 321–326, 2011.

[3] K. R. Basireddy et al. AdaMD: Adaptive Mapping and DVFS for Energy-efficient Heterogeneous Multi-cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[4] L. Benini, A. Bogliolo, and G. De Micheli. A Survey of Design Techniques For System-Level Dynamic Power Management. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 8(3):299–316, 2000.

[5] G. Bhat et al. Algorithmic Optimization of Thermal and Power Management for Heterogeneous Mobile Platforms. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 26(3):544–557, 2018.

[6] G. Bhat et al. Online learning for adaptive optimization of heterogeneous SoCs. In *Proc. of the International Conference on Computer-Aided Design*, page 61, 2018.

[7] G. Bhat, S. Gumussoy, and U. Y. Ogras. Power-temperature stability and safety analysis for multiprocessor systems. *ACM Trans. on Embedded Computing Systems (TECS)*, 16(5s):145, 2017.

[8] G. Bhat, S. Gumussoy, and U. Y. Ogras. Power and thermal analysis of commercial mobile platforms: Experiments and case studies. In *In Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 144–149, 2019.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. Int. Conf. on Parallel Arch. and Compilation Techniques*, pages 72–81, 2008.

[10] P. Bogdan, R. Marculescu, S. Jain, and R. T. Gavila. An Optimal Control Approach to Power Management for Multi-Voltage and Frequency Islands Multiprocessor Platforms under Highly Variable Workloads. In *Proc. of the Intl. Symp. on Networks on Chip*, pages 35–42, 2012.

[11] S. Charles, C. A. Patil, U. Y. Ogras, and P. Mishra. Exploration of Memory and Cluster Modes In Directory-Based Many-Core CMPs. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. IEEE, 2018.

[12] X. Chen et al. Dynamic Voltage and Frequency Scaling For Shared Resources In Multicore Processor Designs. In *Proc. of the Design Autom. Conf.*, page 114, 2013.

[13] Z. Chen and D. Marculescu. Distributed Reinforcement Learning For Power Limited Many-Core System Performance Optimization. In *Proc. of the Design, Automation & Test in Europe Conference & Exhibition*, pages 1521–1526, 2015.

[14] A. Cortex. A15 MPCore Processor Technical Reference Manual. *ARM Holdings PLC*, 24, 2013.

[15] A. K. Coskun, T. S. Rosing, and K. Whisnant. Temperature Aware Task Scheduling in MPSoCs. In *Proc. of the Conf. on Design, Autom. and Test in Europe*, pages 1659–1664, 2007.

[16] J. Deng et al. Imagenet: A large-scale Hierarchical Image Database. In *IEEE conference on computer vision and pattern recognition, pp. 248-255*, 2009.

[17] S. Dey and othters. Edge Cooling Mode: An Agent Based Thermal Management Mechanism For DVFS Enabled Heterogeneous MPSoCs. In *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*, pages 19–24. IEEE, 2019.

[18] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. SPARTA: Runtime Task Allocation For Energy Efficient Heterogeneous Manycores. In *Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10, 2016.

[19] Q. Fettes et al. Dynamic Voltage And Frequency Scaling In NoCs With Supervised And Reinforcement Learning Techniques. *IEEE Trans. Comput.*, 2018.

[20] U. Gupta, J. Campbell, U. Y. Ogras, R. Ayoub, M. Kishinevsky, F. Paterna, and S. Gumussoy. Adaptive Performance Prediction for Integrated GPUs. In *Proc. of the Intl. Conf. on Computer-Aided Design*, page 61, 2016.

[21] U. Gupta et al. Dynamic Power Budgeting For Mobile Systems Running Graphics Workloads. *IEEE Transactions on Multi-Scale Computing Systems*, 4(1):30–40, 2017.

[22] U. Gupta et al. DyPO: Dynamic Pareto-Optimal Configuration Selection For Heterogeneous MpSoCs. *ACM Trans. Embedd. Comput. Syst.*, 16(5s):123, 2017.

[23] U. Gupta et al. Staff: Online learning with stabilized adaptive forgetting factor and feature selection algorithm. In *Proc. of Design Automation Conference (DAC)*, pages 1–6, 2018.

[24] U. Gupta et al. A Deep Q-Learning Approach for Dynamic Management of Heterogeneous Processors. *IEEE Computer Architecture Letters*, 18(1):14–17, 2019.

[25] U. Gupta, S. Korrapati, N. Matturu, and U. Y. Ogras. A Generic Energy Optimization Framework for Heterogeneous Platforms Using Scaling Models. *Microprocessors and Microsystems*, 2016.

[26] M. R. Guthaus et al. Mibench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the Int. Workshop on Workload Characterization*, pages 3–14, 2001.

[27] Hardkernel. ODROID-XU3. `https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3` Accessed 24 Nov. 2018, 2014.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning For Image Recognition. In *Proceedings of The IEEE Conference On Computer Vision And Pattern Recognition*, pages 770–778, 2016.

[29] J. Henkel et al. Dark Silicon: From Computation to Communication. In *Proc. of the Intl. Symp. on Networks-on-Chip*, page 23, 2015.

[30] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In *Proc. of the Int. Symp. on Low Power Elec. and Design*, pages 38–43, 2007.

[31] M. Horro et al. Effect of Distributed Directories in Mesh Interconnects. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 51. ACM, 2019.

[32] C. Isci, G. Contreras, and M. Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems With Application to Dynamic Power Management. In *Proc. of the Intl. Symp. on Microarch.*, pages 359–370, 2006.

[33] Y. Jia et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[34] B. K. Joardar et al. Design and Optimization of Heterogeneous Manycore Systems Enabled By Emerging Interconnect Technologies: Promises and Challenges. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 138–143. IEEE, 2019.

[35] H. K. Jörg Henkel and M. Rapp. Smart Thermal Management for Heterogeneous Multicores. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 132–137. IEEE, 2019.

[36] D. Kadjo, U. Ogras, R. Ayoub, M. Kishinevsky, and P. Gratz. Towards Platform Level Power Management in Mobile Systems. In *Proc. of Intl SoC Conf.*, pages 146–151, 2014.

[37] L. Kernel. Tool, LinuxKernel. `https://en.wikipedia.org/wiki/Perf_(Linux)`, accessed 8 August 2019.

[38] R. G. Kim et al. Wireless NoC and Dynamic VFI Codesign: Energy Efficiency Without Performance Penalty. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 24(7):2488–2501, 2016.

[39] R. G. Kim et al. Imitation Learning For Dynamic VFI Control In Large-Scale Manycore Systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 25(9):2458–2471, 2017.

[40] A. Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *Advances In Neural Information Processing System*, pages 1097–1105, 2012.

[41] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, 2005.

[42] C. Lattner. LLVM and Clang: Next Generation Compiler Technology. In *Proc. of the BSD*, pages 1–2, 2008.

[43] C. Lattner and V. Adve. LLVM: A Compilation Framework For Lifelong Program Analysis & Transformation. In *Proc. of the Int. Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, page 75, 2004.

[44] S. K. Mandal, R. Ayoub, M. Kishinevsky, and U. Y. Ogras. Analytical Performance Models for NoCs with Multiple Priority Traffic Classes. *ACM Trans. on Embedded Computing Systems (TECS)*, 2019.

[45] S. K. Mandal et al. Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[46] 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 Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(1):3–21, 2009.

[47] J. F. Martinez and E. Ipek. Dynamic Multicore Resource Management: A Machine Learning Approach. *IEEE Micro*, 29(5), 2009.

[48] P. Mochel. The Sysfs Filesystem. In *Proc. of the Linux Symp.*, 2005.

[49] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A Portable Interface To Hardware Performance Counters. In *Proc. of the Dept. of defense HPCMP Users Group Conf.*, volume 710, 1999.

[50] T. S. Muthukaruppan et al. Hierarchical Power Management For Asymmetric Multi-Core In Dark Silicon Era. In *Proc. Design Autom. Conf.*, page 174, 2013.

[51] U. Y. Ogras and R. Marculescu. *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*, volume 184. Springer Science & Business Media, 2013.

[52] U. Y. Ogras, R. Marculescu, D. Marculescu, and E. G. Jung. Design and Management of Voltage-Frequency Island Partitioned Networks-on-Chip. *IEEE Trans. on Very Large Scale Integration Systems*, 17(3):330–341, 2009.

[53] V. Pallipadi, S. Li, and A. Belay. Cpuidle: Do Nothing, Efficiently. In *Proc. of the Linux Symp.*, volume 2, pages 119–125, 2007.

[54] V. Pallipadi and A. Starikovskiy. The Ondemand Governor. In *Proc. Linux Symp.*, volume 2, pages 215–230, 2006.

[55] J.-G. Park, N. Dutt, and S.-S. Lim. ML-Gov: A Machine Learning Enhanced Integrated CPU-GPU DVFS Governor For Mobile Gaming. In *Proc. Symp. on Embedd. Syst. for Real-Time Multimedia*, pages 12–21, 2017.

[56] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU Power Management For 3D Mobile Games. In *Design Autom. Conf.*, pages 1–6, 2014.

[57] D. A. Program. Tool, IntelXeonPhi. `http://dap.xeonphi.com/`, accessed 9 August 2019.

[58] B. K. Reddy et al. Inter-cluster Thread-to-core Mapping and DVFS on Heterogeneous Multi-cores. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):369–382, 2018.

[59] S. Ross, G. Gordon, and D. Bagnell. A Reduction of Imitation Learning And Structured Prediction To No-Regret Online Learning. In *Proc. of the Int. Conf. on Art. Intel. and Stat.*, pages 627–635, 2011.

[60] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2014.

[61] S. Schaal. Is Imitation Learning The Route To Humanoid Robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[62] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE micro*, 23(6):84–93, 2003.

[63] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[64] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras. Predictive Dynamic Thermal and Power Management for Heterogeneous Mobile Platforms. In *Proc. of the Conf. on Design, Automation & Test in Europe*, pages 960–965, 2015.

[65] A. Sodani et al. Knights Landing: Second-Generation Intel Xeon Phi Product. 2016.

[66] W. Sun, A. Venkatraman, G. J. Gordon, B. Boots, and J. A. Bagnell. Deeply AggreVaTeD: Differentiable Imitation Learning for Sequential Prediction. In *Proc. 34th Int. Conf. Machine Learning*, volume 70, pages 3309–3318, 2017.

[67] C. Szegedy et al. Going Deeper With Convolutions. In *Proceedings of The IEEE Conference On Computer Vision And Pattern Recognition*, pages 1–9, 2015.

[68] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A case Study From Microscopy Image Analysis. In *2014 IEEE 28th International Parallel And Distributed Processing Symposium*, pages 1063–1072. IEEE, 2014.

[69] S. Thomas et al. CortexSuite: A Synthetic Brain Benchmark Suite. In *IISWC*, pages 76–79, 2014.

[70] TI-INA231. `http://www.ti.com/lit/ds/symlink/ina231.pdf, accessedApril06,2017`.

[71] ul Islam et al. Hybrid DVFS Scheduling For Real-Time Systems Based On Reinforcement Learning. *IEEE Systems J.*, 11(2):931–940, 2017.

[72] N. Vallina-Rodriguez and J. Crowcroft. Energy Management Techniques in Modern Mobile Handsets. *IEEE Comm. Surveys & Tutorials*, 15(1):1–20, 2012.

[73] V. Venkataramani, A. Pathania, M. Shafique, T. Mitra, and J. Henkel. Scalable dynamic task scheduling on adaptive many-core.

[74] E. W. Wächter et al. Predictive Thermal Management for Energy-Efficient Execution of Concurrent Applications on Heterogeneous Multicores. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(6):1404–1415, 2019.

[75] W. Wang, P. Mishra, and S. Ranka. *Dynamic Reconfiguration in Real-Time Systems*. Springer, 2012.

[76] X. Wang et al. A Pareto-Optimal Runtime Power Budgeting Scheme for Many-Core Systems. *Microprocessors and Microsystems*, 46:136–148, 2016.

[77] W. S. Z. L. Xuedan Du, Cai Yinghao. Overview of deep learning. pages 159–164, 2016.

[78] X. Zheng, L. K. John, and A. Gerstlauer. Accurate Phase-level Cross-platform Power and Performance Estimation. In *Proc. of Design Autom. Conf.*, page 4, 2016.