DyPO: Dynamic Pareto-Optimal Configuration Selection for Heterogeneous MpSoCs

UJJWAL GUPTA, Arizona State University
CHETAN ARVIND PATIL, Arizona State University
GANAPATI BHAT, Arizona State University
PRABHAT MISHRA, University of Florida
UMIT Y. OGRAS, Arizona State University

Modern multiprocessor systems-on-chip (MpSoCs) offer tremendous power and performance optimization opportunities by tuning thousands of potential voltage, frequency and core configurations. As the workload phases change at runtime, different configurations may become optimal with respect to power, performance or other metrics. Identifying the optimal configuration at runtime is infeasible due to the large number of workloads and configurations. This paper proposes a novel methodology that can find the Pareto-optimal configurations at runtime as a function of the workload. To achieve this, we perform an extensive offline characterization to find classifiers that map performance counters to optimal configurations. Then, we use these classifiers and performance counters at runtime to choose Pareto-optimal configurations. We evaluate the proposed methodology by maximizing the performance per watt for 18 single- and multi-threaded applications. Our experiments demonstrate an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively.

CCS Concepts: • **Computer systems organization → Multicore architectures**; **System on a chip**; • **Hardware → Platform power issues**;

Additional Key Words and Phrases: Pareto optimization, Energy, Performance per watt, Power, Mobile platforms, Multi-cores, DVFS, DPM, LLVM, Clang, Basic blocks, PAPI, Logistic regression

ACM Reference format:
Ujjwal Gupta, Chetan Arvind Patil, Ganapati Bhat, Prabhat Mishra, and Umit Y. Ogras. 2017. DyPO: Dynamic Pareto-Optimal Configuration Selection for Heterogeneous MpSoCs. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2017), 20 pages.
https://doi.org/https://doi.org/10.1145/3126530

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
This work was supported partially by National Science Foundation (NSF) grants CNS-1526562 and CNS-1526687, and Semiconductor Research Corporation (SRC) task 2721.001.
Author's addresses: U. Gupta, C. Patil, G. Bhat and U. Y. Ogras, School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, 85287; emails: {ujjwal, chetanpatil, gmbhat, umit}@asu.edu; P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainsville, FL, 32611; email: prabhat@ufl.edu.

1539-9087/2017/1-ART1 $15.00
https://doi.org/https://doi.org/10.1145/3126530

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 [8, 34]. The number of power management configurations offered by MpSoCs, such as the number of voltage-frequency levels and active cores, have been increasing steadily to adapt to these dynamically varying requirements. 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. This huge collection results in more than one order of magnitude variation in both power consumption and performance, as shown in Figure 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.

Dynamically selecting the optimal configuration is a challenging task aggravated by two major factors. First, the design space is large for a runtime evaluation and exploration. Therefore, an exhaustive search is prohibitive due to significant overhead associated with exploration. Second, and more importantly, the optimal choice is a strong function of the workload, which itself varies dynamically [4]. For example, bringing the data from memory faster is important upon launching the application, but processing time starts dominating later on. Similarly, the application may go through CPU- and memory-bound phases during its lifetime. Consequently, the optimal configuration changes as the composition of the active applications and their phases vary.

Chip designers and power management architects spend significant effort to attain the optimal power-performance trade-off. For example, Figure 1(a) plots power consumption and execution time of a multi-threaded application for 128 different core and operating frequency configurations. We clearly see 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. 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 [31]. 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 [2, 32]. 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

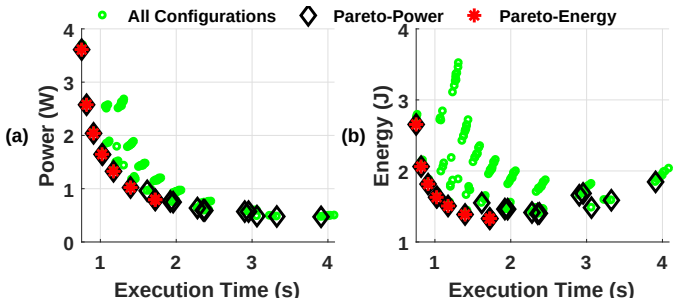

Fig. 1. 128 different frequency and core configurations of the Blackscholes application showing the trade-off between (a) power consumption and execution time, (b) energy consumption and execution time.

energy consumption. For instance, Figure 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. Our experimental results reveal similar trends for default governors for many other metrics, such as performance per watt and instructions per second. Therefore, there is a strong need for runtime algorithms that can choose the optimal configuration with respect to a given metric as a function of the workload.

This paper presents a comprehensive methodology to choose optimal core and frequency configuration at runtime as a function of workload characteristics. Existing approaches rely on core utilizations to make decisions in single steps [31]. In strong contrast, we employ a classifier that chooses the optimal configuration for a given workload phase characterized with a diverse set of performance counters available on the target platform.

Our major contributions towards enabling and validating the proposed methodology are as follows:

- **Instrumentation (Section 3.2):** Finding the optimal configuration as a function of workload is difficult (even offline), since it requires running *precisely the same* workload at each possible configuration. One could run a given application at each possible configuration and collect statistics at uniform time intervals. However, the workload in each time interval would be different for each configuration, since the instructions are processed at different speeds. Therefore, the first step of the proposed methodology is instrumenting the applications using the LLVM [22] compiler infrastructure and PAPI calls [25]. This instrumentation, which has less than 1% overhead, enables us to collect a vast amount of characterization data for each workload snippet[1].
- **Characterization (Section 3.3 & 3.4):** The second step is to collect characterization data using the instrumented applications. In this work, we collected power consumption, processing time and six performance counters for a total of 4,467 workload snippet using 18 different applications. In the third step, we use the power consumption and processing time information to identify the optimal configuration for each of the 4,467 workload snippet with respect to any metric, such as energy, which can be expressed in terms of this information. Finally, the characterization data is used to find classifiers that map each workload snippet to its optimal configuration.
- **Runtime selection (Section 3.5):** Our final step is to develop a new governor that implements the classifier for each metric of interest. The user can easily choose any of the classifiers in this unified governor at runtime by setting a variable at user space. The same features (i.e., performance counters and core utilizations) used for characterization are collected at runtime. Then, the features are fed to the classifier to find the optimal configuration.
- **Experimental validation (Section 4):** We present an extensive set of evaluations using 18 single- and multi-threaded applications running on Odroid XU3. We obtain on average 49%, 45% and 6% lower energy consumption compared to the interactive, ondemand, and powersave governors, respectively. Our approach also outperforms the powersave governor by achieving lower execution time, but has longer execution time than interactive and ondemand governors, as explained in Section 4.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 lays out the groundwork required for collecting meaningful experimental data and the framework for the proposed technique for optimization. Section 4 discusses the experimental results, and Section 5 presents the conclusion.

[1] In this paper, a workload snippet is a sequence of basic blocks with sizes varying from 5k to 100M instructions, as explained in Section 3.2. A group of consecutive snippets make up a workload phase. Each snippet is similar to a micro-benchmark.

2 RELATED RESEARCH

Widespread use of mobile platforms in the last decade is enabled by advanced power management techniques, including dynamic core and uncore scaling [5, 20, 28], cache reconfiguration, task partitioning, task scheduling, and power budgeting [14, 16, 37, 38]. 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 [2]. Similarly, DVFS-based schemes control the operating frequency of a core based on the utilization [17, 26, 31]. For example, millions of commercial mobile platforms run the ondemand and interactive governors [31]. 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 (decrease) the frequency of the processor if the utilization is above (below) a certain threshold [31]. The work presented in [39] 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. In contrast, our approach finds the Pareto-optimal core and frequency configuration in heterogeneous architectures using an extensive set of hardware measurements and multinomial logistic regression. Hence, our approach combines DVFS and DPM by setting the operating frequency/voltage and the type and number of active cores simultaneously.

Recently, a number of studies have focused on workload-aware DPM and DVFS together [1, 6, 10–12, 23, 42]. These techniques choose the best or a mixture of the two strategies to optimize the mobile platform. For instance, the technique proposed in [1] first derives the power and performance models using multivariate linear regression for each different frequency and application. Then, these models are used to determine an optimal performance per watt configuration for an application at runtime. Similarly, the work in [11] proposes an online learning method to select the best-performing DPM policy together with DVFS settings called experts, for a single CPU core. At runtime, the controller characterizes the workload based on energy and cycles-per-instruction models to choose the best-performing expert. The work in [10] proposes a new Linux scheduler to optimize the power consumption under a throughput constraint. Their approach is specifically designed for parallel applications with computation intensive loops. Similarly, the approach proposed in [42] focuses on a group of applications related to web browsing for heterogeneous platforms. They build linear regression models for performance and energy consumption, and then use them to schedule webpages for minimizing the energy consumption of the system. Several recent techniques have also considered applying classification-based methods for the frequency and core selection. For example, the work in [6] proposes a technique for homogeneous server systems, which uses logistic regression to find thread packing and frequency such that the system remains within a power budget. Similarly, the work in [12] uses binning-based classification for identifying the degree of memory- and compute-boundedness of the tasks. Then, these tasks are allocated based on the predicted power and performance to the CPU cores for minimizing the power consumption under a throughput constraint. However, none of the above methods use phase-level instrumentation, which is necessary to identify the optimal configurations for a given workload.

Phase-level performance and power analysis provide a fine grained and reliable information about the workload, as we describe in Section 3.2. This information enables accurate power and performance models across different platforms [41] and practical power management algorithms [18]. 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 [41]. 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 DVFS and DPM approaches, our work leverages the use of phase-level offline characterization for a number of benchmarks to find the Pareto-optimal configurations for each phase. Then, we build classifiers that map the characterized feature data to the Pareto-optimal configurations. Finally, the classifier is used at runtime to select the optimal configuration for a new application phase. In our experimental evaluations, we observe substantial numerical gains in performance per watt compared to a recently proposed algorithm [1] and the default governors.

3 DYPO CONFIGURATION SELECTION

3.1 Motivation and Overview

Modern MpSoCs offer a staggering number of configuration knobs. For example, the recently introduced Samsung Exynos 5422 MpSoC based on ARM big.LITTLE architecture offers four little (A7) and four big (A15) cores that can operate at 13 and 19 different frequencies, respectively [27]. Furthermore, the voltage of each of the core clusters scales with frequency. Since at least one little core has to remain active at all times, this leads to a total of (4×13×4×19) + (4×13) = 4004 different frequency and core configurations. Different configurations lead to a huge variation in power consumption and performance, as shown in Figure 1. Moreover, any given application workload consists of multiple workload phases [33]. 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 [4].

We denote the set of all possible configurations 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, we denote the set of phases encountered during the lifetime of an application by $\mathbf{P}$, and the phase at time k with $p_k \in \mathbf{P}$. Our optimization goal can be expressed as:

$$\begin{aligned} &\textbf{Find } f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C} \\ &\quad \textit{where } c_k^* \in \mathbf{C} \text{ is the optimal configuration} \\ &\quad \textit{for workload phase } p_k \in \mathbf{P} \end{aligned} \tag{1}$$

Identifying the optimal configuration c_k^* at runtime for each phase p_k is a daunting task due to the 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. Clearly, 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 the 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.

Overview and illustrative example: We start with an overview and illustrative example, before detailing the proposed approach. First, we instrument the target application to divide the workload into groups of basic blocks called snippets. This step enables us to collect power and performance statistics of each snippet at runtime, as illustrated in Figure 2. For example, consider an application

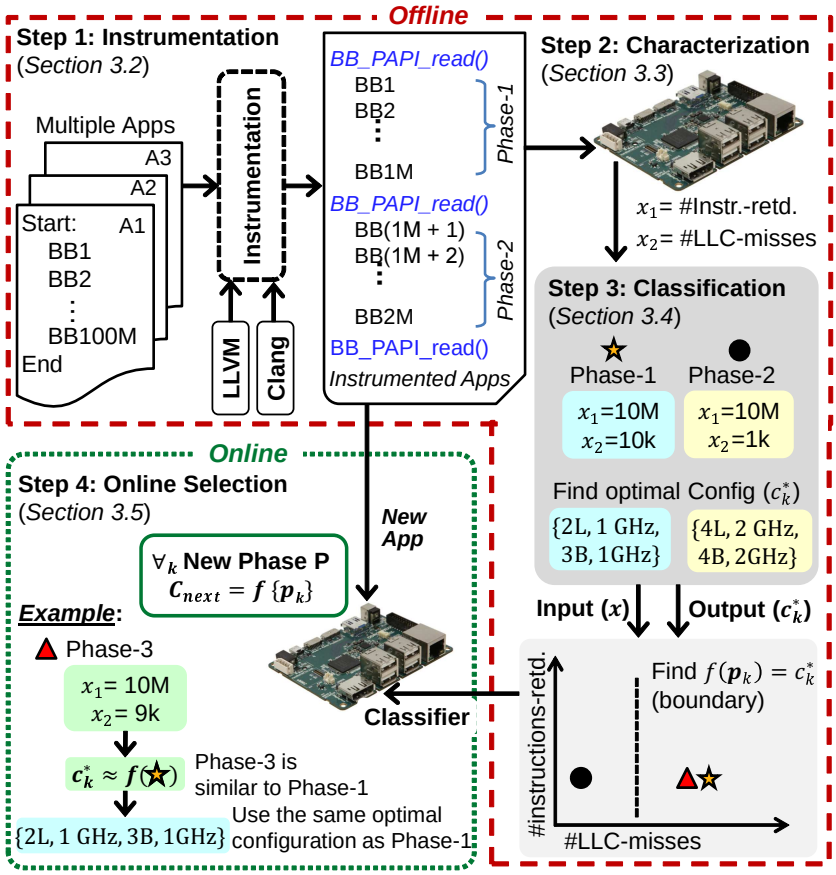

Fig. 2. The outline of the proposed approach with an illustrative example. A block of instructions, such as a function call, makes up basic blocks. Our instrumentation groups a sequence of basic blocks into distinct snippets. Finally, each snippet or a sequence of snippets may form workload phases.

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 BB_PAPI_read() basic blocks that call the PAPI APIs for reading hardware counters and system statistics every 1 million basic blocks. A pair of 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. Offline instrumentation is followed by the characterization step, where we collect extensive power consumption and performance data for a large variety of single- and multi-threaded applications (Section 3.3). More specifically, we collect the data listed in Table 1 while repeatedly running each application using different configurations supported by the platform. Then, this data is used to identify the optimal configuration for each workload snippet. The third step is to design a classifier using this characterization data (Section 3.4). For example, consider two different snippets, 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 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 2 illustrates a potential

Table 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	

classifier that can clearly separate these two snippets. The final step is using the classifier online to determine the optimal configuration for any workload encountered at runtime (Section 3.5). As an example, assume that the system encounters Phase-3, which has 9K *LLC-misses* and 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 our illustrative example is simple, the real problem is multidimensional and far more challenging than creating simple visual boundaries between phases. The rest of this section detail these four steps employed in the proposed methodology.

3.2 Phase-Level Application Instrumentation

Platform designers provide a rich set of hardware and software counters that can be accessed at runtime to identify different workload phases. The PAPI infrastructure provides user level APIs that can be inserted within the application to capture these counters at runtime [25]. In addition to the performance counter information provided by PAPI, it is also important to capture system behavior during the same interval. Therefore, we also log important features, such as the total CPU power consumption, core frequencies, core utilizations, and execution time, by modifying the PAPI API. The system and application level parameters employed in this work are listed in Table 1.

To accurately instrument applications with PAPI APIs, we use the LLVM compiler infrastructure, which has the functionality to analyze any given source code at different granularities, such as module level, function level, and basic block level [22]. 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, we utilize LLVM with *clang* compiler [21] to analyze and instrument PAPI calls at critical basic blocks within an application to collect the hardware counters at runtime.

Figure 3 illustrates the process of instrumenting any benchmark with PAPI calls using LLVM and *clang* compiler. The first step is an instrumentation pass source file in LLVM that can identify existing functions and basic blocks, and add new functions (PAPI APIs) for any application. Then, we use Cmake/Make utilities to compile the LLVM instrumentation pass to get a custom library object file. Finally, we use the *clang* compiler to compile the benchmark with the custom library as an additional input. This generates an output object file that has PAPI APIs instrumented at different basic blocks. Note that our 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.

Instrumenting single-threaded workloads requires identifying the critical basic blocks and then adding simple PAPI calls. While instrumenting the multi-threaded benchmarks, we tie each thread to its own performance counter values. We achieve this with the help of PAPI APIs, which provide

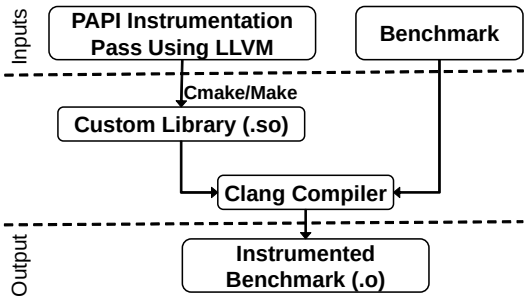

Fig. 3. PAPI API instrumentation overview.

specific calls to register threads that can maintain their own counter data. Since multi-threaded workloads also have phases that only have single threads, we ensure that our instrumentation can capture such phases as well. At the time of logging the hardware counter values along with system performance, we also capture the thread IDs and time-stamp of data collection. This methodology ensures that we are able to analyze both single- and multi-threaded phases of any workload. In practice, inserting PAPI APIs are expected to introduce extra instructions as overhead. Therefore, we ensure that the overhead introduced with these API calls is negligible, as detailed in Section 4.1. Overall, the process of instrumentation enables us to capture the critical regions that provide useful information regarding different phases of an application running on any platform.

3.3 Data Characterization Methodology

Once the benchmarks are instrumented with the PAPI APIs, we collect data for different frequency and core configurations. We first set the highest frequency and core configuration, i.e., 2 GHz for the big cores with all eight cores active. Then, we run three iterations of each benchmark at this frequency and core configuration. Next, we step down the frequency of the big core cluster while maintaining the number of active cores. We repeat this process for each benchmark included in the study. After this, we reduce the frequency level by one, and repeat this process for all supported frequency levels and core configurations. Since the number of total configurations is large even for offline analysis, we use a representative data set obtained by running each benchmark three times with 4×4×8=128 different core and frequency configurations[2]. This selection includes all core configurations (4×4) from 1L+1B to 4L+4B. We include at least one little and one big core, since we are interested in maintaining the heterogeneity of the system. We sweep 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. Indeed, default Android governors also do not utilize lower frequencies. 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}. We run the entire application from start to end for all the selected configurations. Therefore, all the relevant phases are considered irrespective of the application. In this work, our specific knowledge about the target platform is used to choose the frequency configurations. In general, one can also apply formal approaches to select a representative set of configurations [29, 30]. On profiling three iterations of 18 benchmarks for 128 different configurations lead to a total of 6,912 different benchmark runs. We always re-boot the system before starting the data collection process

[2] Time spent for collecting data for 128 configurations on Odroid XU3 is typically about 1-2 hours per benchmark.

Table 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= $n_{\text{big}} \times n_{\text{little}} \times n_{\text{freq}} \times n_{\text{repeat}}$

for each benchmark to ensure consistency of the platform environment. Finally, we collect the characterization data for each workload snippet following the format shown in Table 2.

3.4 Optimal Configuration Classification

After the characterization is complete, we first find the Pareto-optimal configurations for each characterized workload snipped with respect to a given optimization goal. Then, this data combination, i.e., (snippet, optimal configuration) is used to design a classifier. Finally, this classifier is stored on the platform and used at runtime to select the optimal configuration, as detailed in Section 3.5.

3.4.1 ***Optimization Goal.*** Energy consumption and responsiveness are of primary importance in mobile systems [37]. Furthermore, optimizing them also improves performance per watt (PPW). Therefore, we consider a bi-objective optimization problem of minimizing the energy consumption $E(c_k, p_k)$ and execution time $t_{\text{exe}}(c_k, p_k)$ for program snippet p_k and configuration c_k. The optimal cost $J(p_k)$ for this bi-objective problem can be written as follows:

$$J(p_k) = \min_{c_k} \left[E(c_k, p_k) + \mu t_{\text{exe}}(c_k, p_k) \right] \tag{2}$$

where $\mu \geq 0$ is a weight between the energy and execution time that determines the relative importance of the two objectives. For example, when μ is small, the optimization problem essentially turns into minimization of energy (DyPO-Energy), and when μ is large, the optimization problem minimizes the execution time (DyPO-Performance). Any μ value in between will lead to minimizing the energy consumption with some other execution time constraint. *More importantly*, our classification does not depend on the structure of Equation 2, as described next. Therefore, we can compute the Pareto-optimal configurations c_k^* for each snippet p_k for an arbitrary optimization objective that combines energy, execution time, instructions per cycle and power consumption.

3.4.2 ***Design of the Classifier.*** Once the Pareto-optimal configuration c_k^* for each snippet p_k is identified using Equation 2, the next task is to map different snippets to their optimal configurations using the function $f : \mathbf{P} \ni p_k \mapsto c_k^* \in \mathbf{C}$. We utilize multinomial logistic regression classification technique for this purpose due to its simple implementation in the kernel. However, any other supervised machine learning classification technique can be used to the same effect.

To train the logistic regression classifier, we need to use input features and associated output labels, as shown on the upper left corner of Figure 4. The inputs to the classifier are five hardware counters, shown in Table 1 normalized with *instructions-retired*, the sum of the utilizations of the little cores, sorted utilizations of the big cores, and one bias term. The output labels are the optimal configurations found with respect to the criterion in Equation 2. Note that two different snippets can map to the same optimal configuration. Hence, an approach that arbitrarily assigns a supervisory response (optimal configuration) to the features would fail to create a good mapping function *f*. To avoid this, we first employ *k*-means clustering to find natural clustering in the data set [13]. Then, we assign the most frequently occurring optimal configuration in each of the clusters as their output labels. This can also be performed hierarchically with multiple levels of *k*-means clustering and classification. For example, we use two highly accurate classifiers with three classes

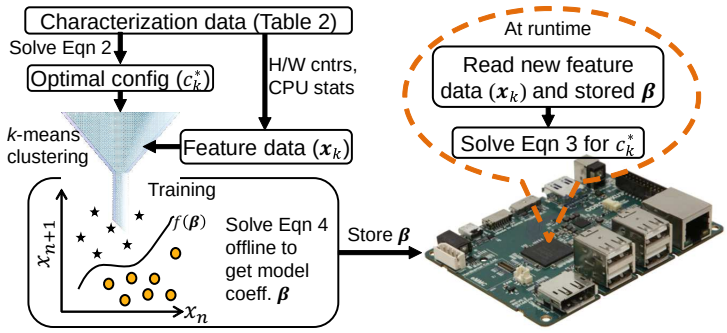

Fig. 4. Training and runtime use of the DyPO classifier.

each in our experiments, as explained in Section 4.2. After the input features and output labels are determined, we design the classifier, as described next.

The conditional probability of the occurrence of a Pareto-optimal configuration $c_k^* \in \mathbf{C}$ given an input $\mathbf{x}_k = [x_1, x_2, x_3, ..., x_N]$, can be represented as $\Pr(\mathbf{C} = c_k^* | \mathbf{x} = \mathbf{x}_k)$. We express the conditional probability for each Pareto-optimal configuration using a logistic function as follows [13]:

$$\Pr(\mathbf{C} = c_k^* | \mathbf{x} = x_k) = \frac{e^{\beta \mathbf{x_k}}}{1 + e^{\beta \mathbf{x_k}}} \tag{3}$$

where $\beta = [\beta_0, \beta_1, ..., \beta_N]$ are the regression coefficients *learned offline* using the characterized data for each workload snippet. The regression coefficients are estimated by maximum likelihood, using the known conditional likelihoods for a class $\mathbf{C}$ given features $\mathbf{x}$ (training data). When the total number of data points (i.e., number of workload snippets × number of configurations) is M, the likelihood function can be written as:

$$\ell(\beta) = \prod_{k=1}^{M} \Pr(\mathbf{C} = c_k^* | \mathbf{X} = x_k) \tag{4}$$

Since the maximum likelihood function in Equation 4 is non-linear, we use the mnrfit function in Matlab to solve for the β values offline. Then, we store the β values as look-up tables in the platform, and use them for selecting the optimal configurations at runtime, as illustrated in Figure 4.

3.5 Online Optimal Configuration Selection

To implement the classifier at the target platform, we need to do *only* the following:

(1) Store the classifier parameters $\beta = [\beta_0, \beta_1, ..., \beta_N]$, where N is the number of input features ($N = 11$ in this work)
(2) Implement Equation 3

At runtime, we read the input features using the PAPI calls for each workload snippet. Then, we plug these features and the β values to Equation 3, as shown in Figure 4. This gives the conditional probability of the occurrence of a Pareto-optimal configuration c_k^* given the input features $\mathbf{x}_k$. Then, the Pareto-optimal configuration c_k^* with the *maximum* conditional probability is selected as the output of the controller.

The proposed approach is highly scalable as it requires only a look-up table for a small number model coefficients β stored in the platform. This occupies very small storage space of only 282 bytes in the Odroid XU3 platform for the 11 features used in our work. Even if we store classifiers for multiple objective functions, such as energy, energy-delay product and performance, the file

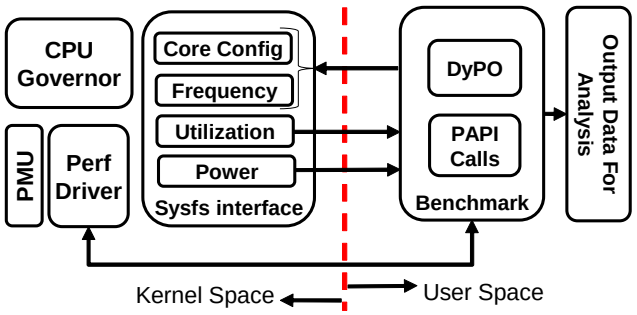

Fig. 5. Implementation of DyPO in Linux Kernel 3.10.

size does not exceed 2 kB. In general, the number of inputs to the classifier are always much smaller than the number of applications, phases and configurations. For example, if the system that needs to be optimized has hundreds of CPU cores, the proposed technique will still require to store only tens of model coefficients for any optimization objective. Note that when the number of features N becomes too large, the cost of computing the logistic function in Equation 3 can increase. In such cases, it is desirable to reduce and select the appropriate number of features using subset selection or Lasso regression [19]. Our approach is also general enough to consider more than two core types. In this case, the characterization data has to include new types of cores. When the number of configurations grow, a subset can be characterized, as detailed in Section 3.3. Since the optimal classifier is designed offline, current offline computing power and existing classification algorithms can easily support solving iterative optimization techniques with tens of types of classes. Finally, the computation complexity of Equation 3 will not increase, making our approach scalable.

4 EXPERIMENTAL RESULTS

This section first describes the experimental setup, including the details of the platform, benchmarks, baseline algorithms and the overhead of our approach. Then, we demonstrate the usefulness of the proposed dynamic Pareto-optimal configuration selection technique by comparing the results of the DyPO-Energy classifier with baseline algorithms and a recently proposed algorithm [1] running on the platform.

4.1 Experimental Setup

We present the experimental results performed on the Odroid XU3 platform running Ubuntu OS with kernel version 3.10 [27]. The platform is equipped with Exynos 5422 chip, which has four little (A7) cores and four big (A15) cores. 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 [36] that report the power consumptions for each CPU cluster, memory and GPU using the I2C driver. We set the sampling frequency of the current sensors to 5 ms to capture small transients in power consumption.

Integration of the DyPO framework with the existing software infrastructure is shown in Figure 5. Our 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 [24]. The Perf driver is mainly responsible

for communicating with the ARM's performance monitoring unit (PMU) [7], which keeps track of different hardware and software counters. We enable the PMU to capture the performance counters listed in Table 1. We also utilize a custom CPU governor to capture per-core utilization through the sysfs interface. The *user space* contains the instrumented benchmarks with PAPI APIs that query the perf driver for performance counters [25]. At runtime, the hardware counters and CPU utilizations at each snippet of the application are used as inputs to the DyPO classifier. The classifier first finds the optimal frequency and core configuration, and then assigns them to the cores using the sysfs interface. We also export time stamps, classifier output and input features to a log file for debugging and offline analysis purposes.

Benchmarks: To validate our implementation, we use eighteen single- and multi-threaded benchmarks from MI-Bench [15], Cortex [35], and PARSEC [3] suites.

Default Governors: The Linux kernel implements a number of 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 frequency [31]. 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. We compare our approach to these three governors[3] because they offer a wide variety of optimization goals and are implemented on millions of smartphones, making them competitive baselines [40].

Overhead Analysis: The DyPO framework induces instrumentation and algorithm runtime overheads. 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 APIs. The baseline is the case when no APIs are inserted within the benchmark. As opposed to the baseline, the APIs in our approach have to be added in the source code to form different workload snippets, as explained in Section 3.2. We observe a very low mean and median overheads of 1.0% and 0.2% across all the 18 different benchmarks used in this paper. The overhead of our runtime selection algorithm is 20μs, 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 our approach 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. Our algorithm is called in the same way as the default frequency governor. As shown in Figure 5, the DyPO approach is implemented within the application to enable phase-level analysis. Therefore, during the decision process of the classifier, a single-threaded application pauses for 20μs. For multi-threaded applications, only one thread has to be paused for 20μs, other threads are not paused and continue to run normally.

4.2 Classifier Accuracy

We use two classifiers in a hierarchical fashion, as explained in Section 3.5. The first classifier is a Level-1 classifier that outputs three probabilities. The highest probability class is chosen as the output of the classifier. Out of the three classes, two lead to specific frequency and core configurations. The third class fires another classifier, which we call the Level-2 classifier. The Level-2 classifier also outputs three classes that lead to specific frequency and core configurations.

The entire data set is divided into 60% training-validation set and 40% for test set on the actual platform. We train the classifiers using the training-validation set. Then, we use the classifiers at runtime for the entire data set (see Section 4.3 for results). Figure 6 shows the accuracy of both classifiers for the training-validation set. The accuracy for the Level-1 classifier across all the benchmarks is very high, with an average of 99.9%. The average accuracy of the Level-2 classifier

[3]We kept the default cpuidle [32] governor active for the frequency governors to enable changes in the core configuration.

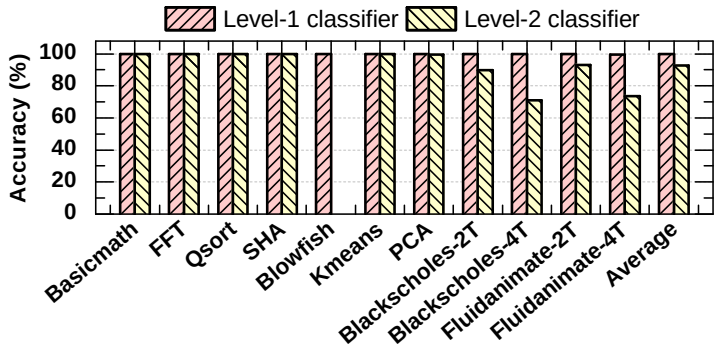

Fig. 6. Accuracy of the two classifiers used on the Odroid platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively.

for the benchmarks is 92.7%. The Blowfish benchmark never uses the Level-2 classifier, i.e., all of its snippets map to the Level-1 classifier only. Single-threaded applications achieve close to 100% accuracy for Level-2 classifier. However, the multi-threaded applications do not perform as well as the single-threaded benchmarks across all the three classes in the Level-2 classifier. For example, Blackscholes-4T shows 71% accuracy as opposed to 100% accuracy of the Basicmath application. This is because all the features of Blackscholes-4T are close to each other and harder to separate into different classes at the second level. We also assess the robustness of the classifiers to unknown data inputs by applying 5-fold cross-validation on the training-validation set. Our results for the 5-fold cross-validation show high average accuracy of 99.9% and 80.5% for the Level-1 and Level-2 classifiers, respectively.

4.3 Runtime Validation of DyPO

In this section, we present the validation of the proposed dynamic Pareto-optimal configuration selection approach by using the DyPO classifier at runtime. We use DyPO-Energy for illustration, since energy minimization is one of the main objectives in mobile platforms. At runtime, DyPO reads the hardware counters and utilization during each workload snippet as inputs to the classifier. Then, the classifier computes the probabilities of the optimal configurations using Equation 2. Finally, the configuration with the highest probability is assigned to the system for the next.

Figure 7 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 (×), and the proposed DyPO-Energy approach (△). 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 7(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 the energy consumption. In contrast, the DyPO-Energy approach runs the application at the lowest energy point of the Pareto frontier at about 14 seconds 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 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 clearly other configurations in the system that could have achieved lower energy consumption for the same execution time. The rest of the plots in Figure 7(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 the 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, we analyze their energy consumption and performance trade-off in Figure 7(o-r). In particular, Figure 7(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 our 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. We observe that 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 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 our technique achieves lower energy than the default governors, as illustrated in Figures 7(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 number of active cores listed in Table 1. Therefore, when other background applications are running, the load perceived by the governor changes. For example, the background applications can increase the CPU utilizations, 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. In fact, there were always hundreds of Linux OS background applications when we performed our experiments. To demonstrate the operation with multiple applications more explicitly, we simultaneously executed two applications, Basicmath (in foreground), and Patricia (in background). Figure 7(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. We also observe 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

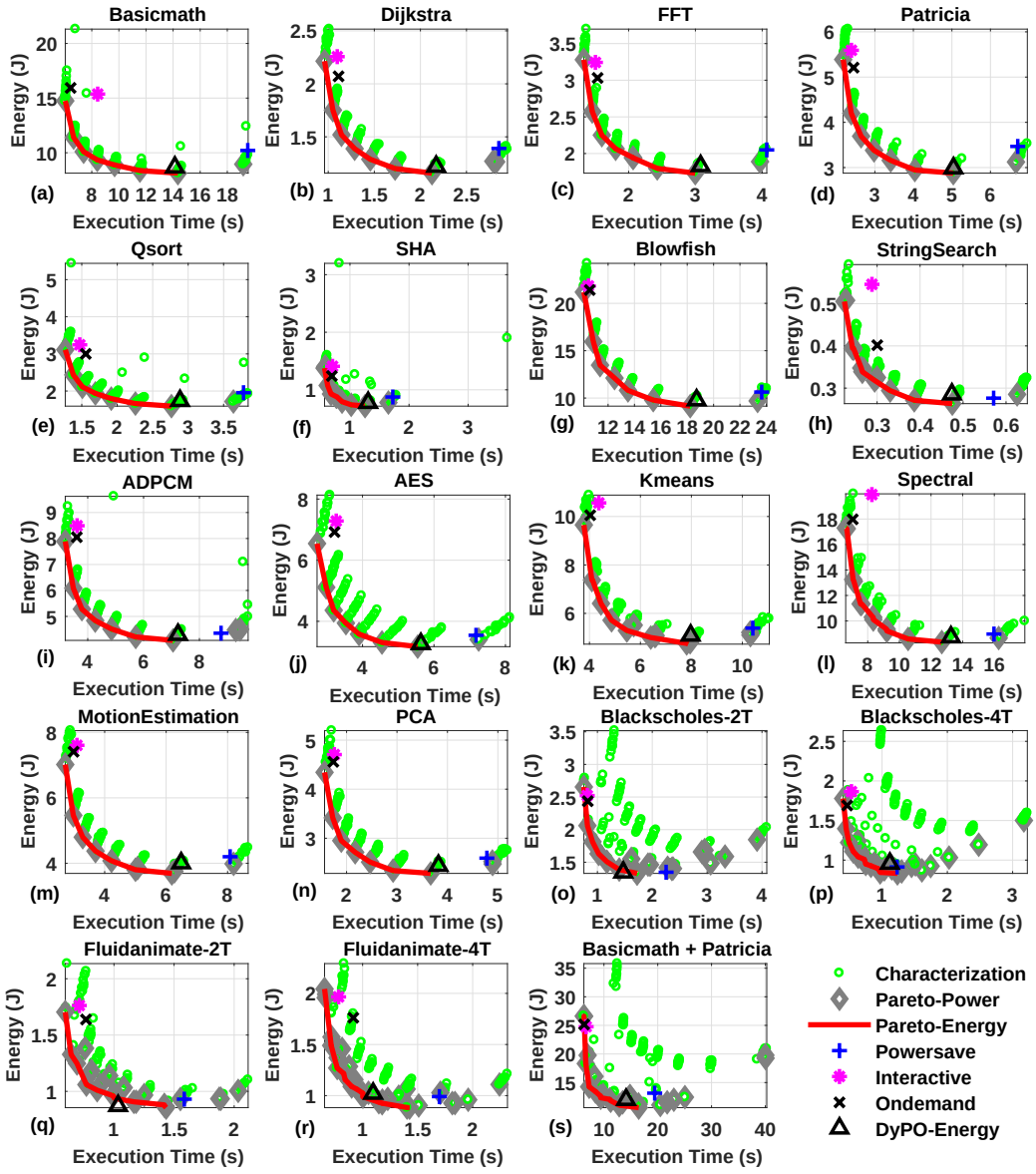

Fig. 7. DyPO-Energy approach compared with the default governors running on the platform. In multi-threaded benchmarks, -2T and -4T represents two and four threads, respectively.

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 7(a) and (d) equal to 11.7 J (sum of 8.7 J and 3 J). This further corroborates our claim that multiple applications can be optimized by using the DyPO-Energy approach effectively.

Note that we can choose any optimization objective in the DyPO technique, such as maximizing performance, minimizing energy with execution time constraint, minimizing the energy-delay product, as mentioned in Section 3.4. For example, we also experimented on performance (DyPO-Performance), in which case our framework always chose the highest points on the Pareto frontier (lowest execution time). This matches closely with the performance governor in the platform that is designed to achieve maximum performance. Also, the DyPO-PPW (maximizing performance per watt) results are similar to DyPO-Energy in our setup, since the number of instructions are almost same for a given application run due to phase-level instrumentation.

4.4 Improvements in Energy and PPW

This section summarizes the advantages of the proposed methodology with respect to the default governors for each benchmark. To this end, we normalize the energy consumption, power consumption, execution time and PPW obtained for each governor with DyPO-Energy results. For example, Figure 8 shows the normalized energy consumption of all the benchmarks compared with the interactive, ondemand and powersave governors. We observe 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. Furthermore, this comes at the expense of almost 24% increase in execution time, as shown in Figure 9. The power consumed by the interactive and ondemand governors is about 3.5× that of the DyPO-Energy, as shown in Figure 10, while the power consumed by the powersave governor is about 23% lower. We also observe that the DyPO-Energy provides 93%, 81%, 6% more PPW than interactive, ondemand, and powersave governors, respectively (shown in Figure 11). Note that compared to the powersave governor, DyPO-Energy provides both energy savings and higher performance. When compared to the ondemand and interactive governors DyPO-Energy obtains substantial reductions in energy consumption albeit with lower performance, as shown in Figure 9. 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. Each model is unique for an application and frequency level. That is, there are as many power and performance models as the number of supported frequencies in the platform for each application. These models are used for computing the PPW for all the supported frequencies and core configurations for a given application. There are two methods to their operation to maximize PPW at runtime. The first is offline (Aalsaud-offline) where the power consumption and performance models associated with an application are pre-characterized. The optimal configuration is found at runtime by a simple linear search through all possible frequency and core configurations. The second method is adaptive (Aalsaud-ADA) that works for an uncharacterized application. That is, an application for which the models are not known. Therefore, they determine the power and performance models at runtime for the adaptive method. To achieve this, they first sweep the frequency every 200 ms. In each 200 ms interval, they measure power and IPC data for at least three different core configurations. Then, they apply linear regression on this data to find the models. Clearly, this is an overhead, since the system runs at non-optimal configuration for 200 ms times the number of frequency levels. However, this happens only one time, once the application is learned,

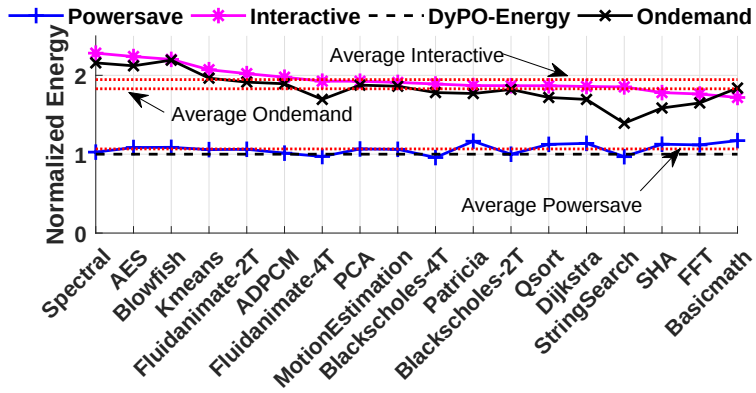

Fig. 8. DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized energy consumption.

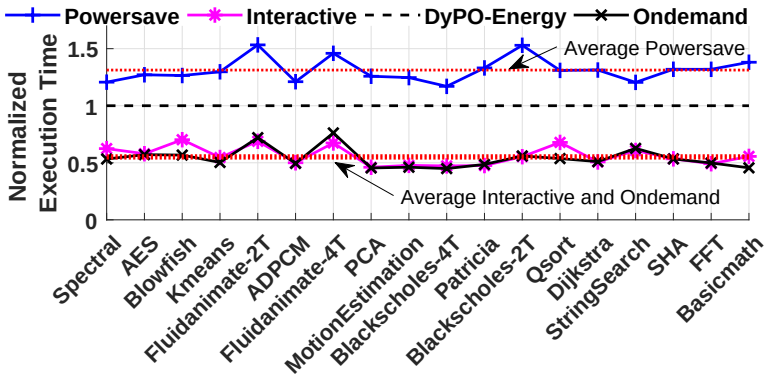

Fig. 9. DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized execution time.

the model is saved in a file for future use. Unlike the proposed approach, Aalsaud et al. [1] profiles the system at fixed time intervals. Since the PAPI APIs are not built to sample an application based on time, we used the perf utility [9] in the Odroid XU3 board to profile the applications every 50ms.

Figure 12 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 four little and big cores. This wastes the extra energy headroom, whereas the ondemand governor utilizes it by keeping the frequency below 1 GHz. We see similar behavior 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.

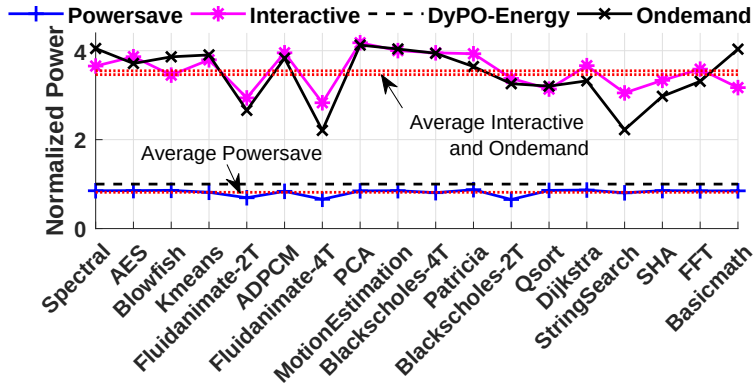

Fig. 10. DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized power consumption.

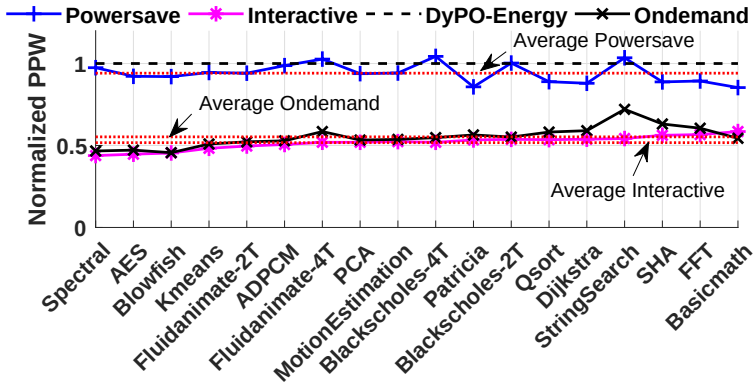

Fig. 11. DyPO-Energy, Interactive, Ondemand and Powersave governor comparison for normalized PPW.

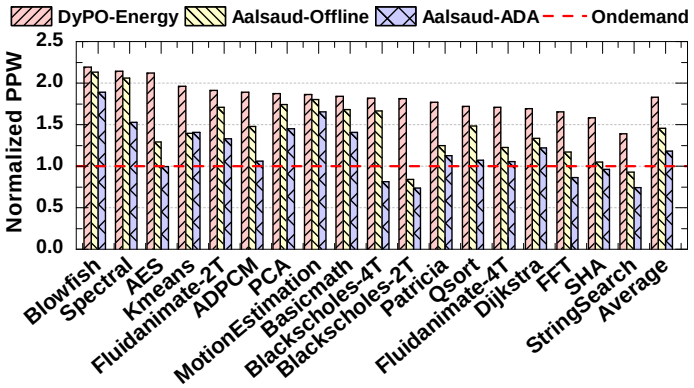

Fig. 12. Comparison of the normalized PPW obtained using DyPO-Energy approach and Aalsaud et al. [1].

5 CONCLUSION

Continued demand for performance led to powerful mobile platforms with heterogeneous multi-processor system on chips. These platforms provide many voltage-frequency levels and active core configurations that can be chosen at runtime. This paper 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. First, phase-level offline characterization for a number of benchmarks is performed to find the Pareto-optimal configurations for each workload snippet. Then, classifiers that map the characterized data to the Pareto-optimal configuration are learned offline using multinomial logistic regression. Finally, the classifiers are used at runtime to select the optimal configuration with respect to a specific metric, such as energy consumption. Our experiments show an average increase of 93%, 81% and 6% in performance per watt compared to the interactive, ondemand and powersave governors, respectively.

REFERENCES

[1] A. Aalsaud *et al.*, "Power–Aware Performance Adaptation of Concurrent Applications in Heterogeneous Many-Core Systems," in *Proc. of the Intl. Symp. on Low Power Elec. and Design*, 2016, pp. 368–373.

[2] 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.*, vol. 8, no. 3, pp. 299–316, 2000.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of the Intl. Conf. on Parallel Arch. and Compilation Tech.*, 2008, pp. 72–81.

[4] 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*, 2012, pp. 35–42.

[5] X. Chen *et al.*, "Dynamic Voltage and Frequency Scaling for Shared Resources in Multicore Processor Designs," in *Proc. of the Design Autom. Conf.*, 2013, p. 114.

[6] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps," in *Proc. of the Intl. Symp. on Microarch.*, 2011, pp. 175–185.

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

[8] 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*, 2007, pp. 1659–1664.

[9] A. C. de Melo, "The New Linux Perf Tools," in *Linux Kongress*, vol. 18, 2010.

[10] E. Del Sozzo *et al.*, "Workload-aware Power Optimization Strategy for Asymmetric Multiprocessors," in *Proc. of the Design, Auto. & Test in Europe Conf. & Exhib.*, 2016, pp. 531–534.

[11] G. Dhiman and T. S. Rosing, "System-Level Power Management Using Online Learning," *IEEE Trans. Comput.-Aided Design Integr. Circuits and Syst.*, vol. 28, no. 5, pp. 676–689, 2009.

[12] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores," in *Proc. of the Intl. Conf. on Hardware/Software Codesign and Sys. Syn.*, 2016, p. 27.

[13] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*. Springer Series in Statistics, Berlin, 2001, vol. 1.

[14] U. Gupta *et al.*, "Dynamic Power Budgeting for Mobile Systems Running Graphics Workloads," *IEEE Trans. on Multi-Scale Comp. Sys.*, 2017.

[15] M. R. Guthaus *et al.*, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proc. of the Intl. Work. on Workload Char.*, 2001, pp. 3–14.

[16] J. Henkel *et al.*, "Dark Silicon: From Computation to Communication," in *Proc. of the Intl. Symp. on Networks-on-Chip*, 2015, p. 23.

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

[18] 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.*, 2006, pp. 359–370.

[19] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*. Springer, 2013, vol. 6.

[20] R. G. Kim *et al.*, "Wireless NoC and Dynamic VFI Codesign: Energy Efficiency Without Performance Penalty," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 7, pp. 2488–2501, 2016.

[21] C. Lattner, "LLVM and Clang: Next Generation Compiler Technology," in *Proc. of the BSD*, 2008, pp. 1–2.

[22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of the Intl. Symp. on Code Gen. and Opt.: Feedback-directed and Runtime Opt.*, 2004, p. 75.
[23] J. Li and J. F. Martinez, "Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors," in *Proc. of the Intl. Symp. on High-Perf. Comp. Arch.*, 2006, pp. 77–87.
[24] P. Mochel, "The Sysfs Filesystem," in *Proc. of the Linux Symp.*, 2005.
[25] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *Proc. of the Department of Defense HPCMP Users Group Conf.*, 1999.
[26] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era," in *Proc. of the Design Autom. Conf.*, 2013, pp. 1–9.
[27] ODROID. Platforms, ODROID – XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825, accessed 6 April 2017.
[28] U. Y. Ogras and R. Marculescu, *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*. Springer Science & Business Media, 2013, vol. 184.
[29] G. Palermo, C. Silvano, and V. Zaccaria, "Multi-objective Design Space Exploration of Embedded Systems," *Jrnl of Embd. Comp.*, vol. 1.3, pp. 305–316, 2005.
[30] M. Palesi and T. Givargis, "Multi-objective Design Space Exploration Using Genetic Algorithms," in *Proc. of the Intl. Symp. on Hardware/Software Codesign*, 2002, pp. 67–72.
[31] V. Pallipadi and A. Starikovskiy, "The Ondemand Governor," in *Proc. of the Linux Symp.*, vol. 2, 2006.
[32] V. Pallipadi, S. Li, and A. Belay, "Cpuidle: Do Nothing, Efficiently," in *Proc. of the Linux Symp.*, vol. 2, 2007, pp. 119–125.
[33] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and Exploiting Program Phases," *IEEE micro*, vol. 23, no. 6, pp. 84–93, 2003.
[34] 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*, 2015, pp. 960–965.
[35] S. Thomas *et al.*, "CortexSuite: A Synthetic Brain Benchmark Suite," in *Proc. of the Intl. Symp. on Workload Char.*, 2014, pp. 76–79.
[36] TI-INA231. http://www.ti.com/lit/ds/symlink/ina231.pdf, accessed April 06, 2017.
[37] N. Vallina-Rodriguez and J. Crowcroft, "Energy Management Techniques in Modern Mobile Handsets," *IEEE Comm. Surveys & Tutorials*, no. 99, pp. 1–20, 2012.
[38] W. Wang, P. Mishr A, and S. Ranka, *Dynamic Reconfiguration in Real-Time Systems*. Springer, 2012.
[39] X. Wang *et al.*, "A Pareto-Optimal Runtime Power Budgeting Scheme for Many-Core Systems," *Microprocessors and Microsystems*, vol. 46, pp. 136–148, 2016.
[40] XDA-Developers Forums. https://forum.xda-developers.com/general/general/ref-to-date-guide-cpu-governors-o-t3048957, accessed April 06, 2017.
[41] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate Phase-level Cross-platform Power and Performance Estimation," in *Proc. of Design Autom. Conf.*, 2016, p. 4.
[42] Y. Zhu and V. J. Reddi, "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems," in *Intl. Symp. on High Perf. Comput. Arch.*, 2013.

Received April 2017; revised June 2017; accepted June 2017