SLOOP: QoS-Supervised Loop Execution to Reduce Energy on Heterogeneous Architectures

M. WAQAR AZHAR, PER STENSTRÖM, and VASSILIS PAPAEFSTATHIOU,
Chalmers University of Technology

Most systems allocate computational resources to each executing task without any actual knowledge of the application's Quality-of-Service (QoS) requirements. Such best-effort policies lead to overprovisioning of the resources and increase energy loss. This work assumes applications with soft QoS requirements and exploits the inherent timing slack to minimize the allocated computational resources to reduce energy consumption.

We propose a lightweight progress-tracking methodology based on the outer loops of application kernels. It builds on online history and uses it to estimate the total execution time. The prediction of the execution time and the QoS requirements are then used to schedule the application on a heterogeneous architecture with big out-of-order cores and small (LITTLE) in-order cores and select the minimum operating frequency, using DVFS, that meets the deadline. Our scheme is effective in exploiting the timing slack of each application. We show that it can reduce the energy consumption by more than 20% without missing any computational deadlines.

CCS Concepts: • **Computer systems organization → Architectures**; **Embedded systems**; • **Hardware → Power and energy**;

Additional Key Words and Phrases: Energy, heterogeneous architecture, quality of service, big-LITTLE, streaming applications, soft real-time

ACM Reference format:
M. Waqar Azhar, Per Stenström, and Vassilis Papaefstathiou. 2017. SLOOP: QoS-Supervised Loop Execution to Reduce Energy on Heterogeneous Architectures. *ACM Trans. Archit. Code Optim.* 14, 4, Article 41 (December 2017), 25 pages.
https://doi.org/10.1145/3148053

1 INTRODUCTION

The end of Dennard scaling has made power consumption of microprocessor chips a major challenge. The current technological limitations impose significant power constraints on new designs and restrain the computational growth of next-generation systems. In addition, reducing energy consumption of mobile systems remains important because of battery capacity and size/weight limitations. Therefore, efficient resource allocation is paramount to minimize overprovisioning of resources and to save energy. Initial works on reducing energy consumption targeted *dynamic*

This research was supported by grants from the Swedish Research Council (contract number 2012-4924) and the European Research Council (ERC) under the MECCA project (contract 340328). The simulations were run on the resources provided by the Swedish National Infrastructure for Computing (SNIC) at C3SE.
Authors' addresses: M. W. Azhar, P. Stenström, and V. Papaefstathiou, Department of Computer Science and Engineering, Chalmers University of Technology, 41296, Gothenburg, Sweden; emails: {waqarm, per.stenstrom, vaspap}@chalmers.se.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
2017 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 1544-3566/2017/12-ART41 $15.00
https://doi.org/10.1145/3148053

voltage-frequency scaling (DVFS), where the objective is to find the lowest voltage-frequency pair that maintains the same level of performance across computational phases despite variations in instruction-level parallelism (ILP) and memory-level parallelism (MLP). A limited amount of available variation in ILP and/or MLP, however, limits the amount of energy savings obtained by DVFS.

Another popular approach is to use *Power Gating (PG)*, where some parts of or the entire processor is powered down during idle periods, still delivering the same average performance. However, in order to save more energy than DVFS, the idle periods must be long enough to compensate for the cubic reduction of power in relation to frequency offered by DVFS. One can make idle periods longer using techniques like *Computational Sprinting* (Raghavan et al. 2012). However, the power consumption during the active phase will be higher due to higher frequency and voltage, which, again, is cubic with respect to frequency (Själander et al. 2014; Kaxiras and Martonosi 2008).

On the other hand, multicore *processor heterogeneity* (Kumar et al. 2003) is an effective and orthogonal approach to DVFS to save energy that has been increasingly adopted by many mobile chips. Such chips feature different types of computing elements, each with different performance/energy tradeoffs (e.g., ARM's big-LITTLE consists of big out-of-order cores and LITTLE in-order cores, GPUs, and/or custom accelerators). Scheduling tasks on the set of heterogeneous computing elements provided by a multicore system to meet performance and energy goals has attracted a lot of attention (Själander et al. 2014). However, the goal of most prior work has been to offer *constant average performance* rather than to reduce power and energy consumption. By contrast, Suh et al. (2015) present an approach that regulates the Million-Instructions-Per-Second (MIPS) rate using DVFS to meet the computational deadline to save energy. Unfortunately, they target an offline calculated MIPS rate, which is determined using the worst-case number of executed instructions. This estimation is inherently pessimistic and leads to overprovisioning of resources. In addition, they do not exploit processor heterogeneity.

The goal of this article is to take into account "soft" QoS requirements in an application-agnostic manner and use both DVFS and heterogeneity to save energy. The rationale is that several classes of applications are able to tolerate *low performance* as long as the deadlines imposed by their QoS requirements are met. The notion of QoS and computational deadline offers a big opportunity to save energy by reducing the amount of resources allocated to each task needed to meet its deadline. The intuition behind our proposed *QoS-Based Resource Management* is that systems typically execute the tasks faster than needed to meet their QoS, and in this process, they generate timing *slack*. Our thesis is that we can save energy by exploiting this slack by both slowing down the application (by exploiting DVFS) and rescheduling it on a slower but more energy-effective core (by exploiting processor heterogeneity) and complete it at, or just before, the deadline.

We propose online prediction of the number of instructions to be executed in the next phase coupled with a supervisory mechanism based on slack to achieve just-in-time completion of each computational phase. Our approach instruments the outer loops of the application (which often constitute the bulk of the execution) and dynamically monitors the characteristics of a loop, in terms of number of instructions and cycles per iteration, inside a runtime layer via existing hardware performance counters. The runtime layer predicts the number of instructions and the time for the whole execution of the program (or a part of it) based on the history and performs *supervised loop execution (SLOOP)* by controlling DVFS and selecting a core type (big out-of-order (OoO) vs. LITTLE in-order core) that meets the QoS deadline.

Specifically, the contributions of this article are as follows:

(1) A lightweight runtime progress-tracking methodology based on the execution of outer loops in the applications
(2) Simple, yet accurate, instruction and execution-time predictors based on instruction-count history

(3) A novel scheduler for heterogeneous multicores that saves energy by tuning the frequency and the core type based on QoS, the prediction, and the available slack

We have implemented the scheduler and also present an evaluation of the overall proposed methodology run on a real ARM-based big-LITTLE system to quantify the potential of using QoS to save energy by DVFS and core-type scheduling. We show that our method can save 23% of energy on average for the tightest QoS setting. Energy savings can further improve to 62% when choosing a deadline that is twice that of the baseline case.

The rest of the article is organized as follows. Section 2 provides further motivation of our approach. Then, Section 3 presents our proposed SLOOP scheme for runtime progress tracking. We move on to the evaluation by presenting the experimental methodology in Section 4 and the results in Section 5. Section 6 then positions our contributions in relation to prior art before we conclude in Section 7.

2 MOTIVATION

Streaming applications (e.g., in multimedia) spend the majority of their time inside loops with multiple iterations, where each iteration typically does the computation on an object (e.g., a frame). First, this iterative behavior can be utilized to predict the execution time of future iterations. Second, iteration boundaries provide a natural granularity for monitoring and scheduling, compared to fixed-time intervals that have been proposed in other studies (e.g., Craeynest et al. (2012), Suh et al. (2015), and Su et al. (2014)). Computations in successive iterations are logically related yet use a new set of data. Thus, scheduling decisions (e.g., switching between cores) taken at iteration boundaries can minimize the performance penalties because of data movement. For this reason, our approach is to monitor the behavior of past iterations to predict the behavior of the next iterations.

Using a multimedia application as an example, it is well known that while there is some execution-time correlation between iterations, because the data being computed in consecutive iterations is related, there is also variability in the execution time across iterations (Hughes et al. 2001a). This variability stems from differences in control flow and data locality. This variability can, however, be put to good use. Our thesis is that, since there is correlation in data across loop iterations, the variability in execution time can be predicted. Our goal is to use execution-time prediction of future iterations so as to adapt the underlying computational resources (DVFS, core type) to closely meet the deadlines in order to save energy. In the subsequent sections, we provide motivational data for the viability of this approach.

2.1 Execution-Time Variability

In order to assess the degree of variability in application loops, we use a set of multimedia/graphics benchmarks from ALPBench (Li et al. 2005) and SPEC2006 (Henning 2006). We study the execution time per iteration on a heterogeneous ARM big.LITTLE architecture described in detail in Section 4. In these benchmarks, we do online profiling of the iterations of the outer loops where each iteration performs the computations required to generate a multimedia/graphics frame.

Figure 1 presents the cumulative distribution function (CDF) of an iteration's execution time T for the set benchmarks used. The x-axis shows the execution time of iterations normalized to the slowest iteration found in the specific application run. The y-axis shows the cumulative percentage of the frame instances that have execution times less than or equal to the x value. For instance, the CDF for mpegenc1 in Figure 1(a) shows that around 30% of the frames (y-axis) complete with an execution time lower than or equal to 65% (x-axis) of the slowest executing frame.

The plots show that, for several applications, a considerable number of iterations execute faster than the slowest iteration. For instance, the CDF of the application RayTrace_teapot in Figure 1(a) shows that 60% of the iterations complete faster or at a time corresponding to 65% of the slowest

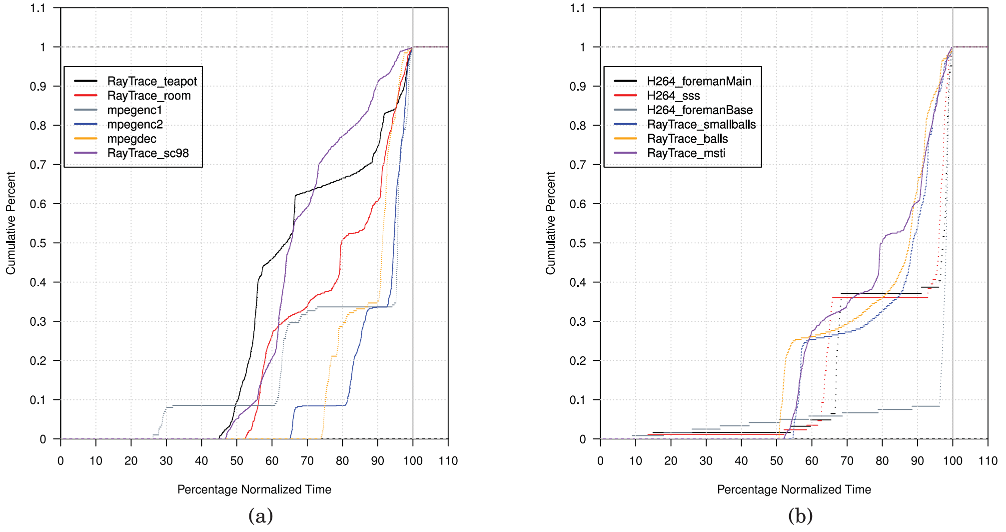


Fig. 1. The Cumulative Distribution Function (CDF) of frame execution times for three benchmarks (Raytrace, MPEG encoder, and H264 encoder) on a number of input sets.

iteration instance. Given a deadline, execution of all the iterations can be adjusted to it so that each iteration completes at roughly the same time and close to the deadline. This analysis shows that there is an opportunity to exploit the inherent variability across frames to save energy by allowing a frame to execute slower, using DVFS or a more energy-efficient core type, without missing its deadline.

Another important aspect of interiteration variability is the magnitude of the variability over time. Figure 2 (top) shows the execution time per iteration in the Raytrace benchmark with the input set teapot. We observe that the per-iteration execution time increases or decreases by a small amount across iterations, but during the whole execution there are spikes, upward or downward, that last for a few tens of iterations. In order to quantify the percentage of interiteration execution-time change as a function of time, Figure 2 (bottom) is shown. Here, we note that the iteration-to-iteration changes are very small and do not exceed 3%. This behavior appears in most of the benchmarks we study and, as we will see, can be easily predicted.

Table 1 shows various statistics for all benchmarks, such as average, maximum, minimum, standard deviation, relative standard deviation, and relative change of execution time per iteration. Relative change is calculated as $(T_{max} - T_{min}) * 100/T_{max}$. Here it can be seen that there is a considerable amount of variability across the iteration execution times in each benchmark that can be exploited to reduce energy. If the computational deadline is the same as the maximum execution time, the relative change indicates how much the fastest iteration can be slowed down to save energy. As we can see, relative change is at minimum 23% for ALP → MPEG Decoder → blah benchmark but can be as high as 91%. Section 5.3 shows that it can lead to considerable energy savings.

The predictable interiteration behavior allows us to accurately predict the execution time of iterations in the near future. This opens up an opportunity to select a slower and more energy-efficient core (processor heterogeneity) and a lower operating frequency (DVFS) when there is enough *time slack* or select a high-performance core and a higher operating frequency when there is a danger to miss deadlines. Our goal is to select the most energy-efficient power state (DVFS and core type) that meets the QoS requirement to save as much energy as possible.

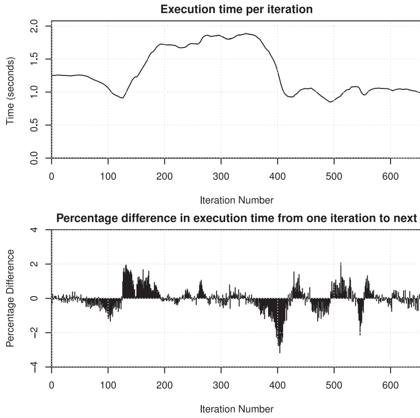


Fig. 2. The execution time per iteration as a function of time (top diagram) and percentage changes of execution time from one iteration to the next iteration as a function of time (bottom diagram) for the Raytrace application with input teapot.

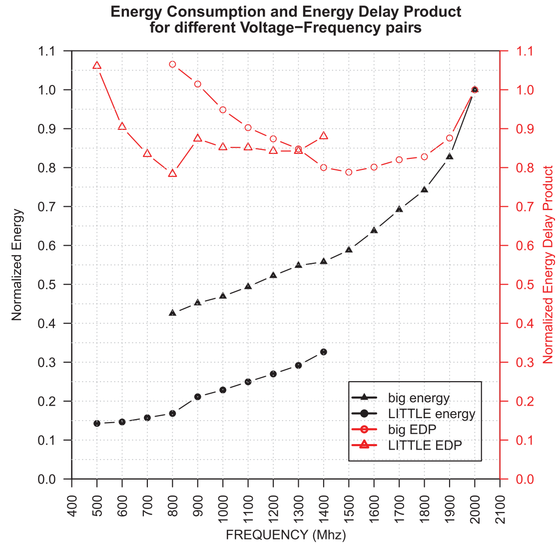


Fig. 3. Energy consumption and EDP under different frequency-voltage and core types for Raytrace application with input teapot.

Table 1. Minimum, Maximum, Mean, Standard Deviation, Relative Standard Deviation, and Relative Change of Execution Times Per Loop Iteration

Workload	Average Time (msec)	Maximum Time (msec)	Minimum Time (msec)	Standard Deviation (msec)	Relative Standard Deviation (%)	Relative Change (%)
ALP → Raytrace → teapot	1,312	1,890	848	344	26	55
ALP → Raytrace → balls	1,030	1,308	662	225	22	49
ALP → Raytrace → room	436	556	292	89	20	47
ALP → Raytrace → 2balls	252	291	208	31	12	29
ALP → Raytrace → smallballs	603	740	404	118	20	45
ALP → Raytrace → sc98	864	1,218	569	158	19	53
ALP → Raytrace → msti	986	1,259	655	203	21	48
ALP → MPEG Encoder → blah	146	177	46	39	26	74
ALP → MPEG Encoder → teapot	254	281	182	26	10	35
ALP → MPEG Decoder → blah	11	13	10	1.04	9	23
SPEC 2006 → H264 Encoder → foreman base	1,419	1,512	136	239	17	91
SPEC 2006 → H264 Encoder → foreman main	1,749	2,049	305	366	21	85
SPEC 2006 → H264 Encoder → sss	11,806	14,002	1,882	2,496	21	87

2.2 Energy Levels in Heterogeneous Architectures

To concretely establish the potential energy savings of heterogeneity combined with DVFS, we review the power states offered by the ARM system we use in our investigation. Although running slower with lower frequencies or more energy-efficient cores saves power, we are also interested in assessing the potential of reducing the *energy-to-solution*. When expanding in time with the

same amount of powered-up resources, then static energy could dominate; this is a typical case in homogeneous platforms with big cores. However, we consider heterogeneous architectures like ARM big.LITTLE and measure the associated energy-to-solution using the platform described in Section 4. Figure 3 shows the total processor energy consumption (energy-to-solution) for the execution of the RayTrace benchmark from ALP with input teapot across different frequencies (DVFS settings) and on big and LITTLE cores. These measurements support our claim for saving energy by running slower on heterogeneous architectures. Moreover, they show that the energy savings can be substantial. We also show the energy delay product (EDP) for all of the cases. EDP is presented as normalized to the EDP of the base case (i.e., 2GHz on a big core). EDP for most power states is less than the base EDP, with a few power states showing a small increase with a maximum value of around 6%. This indicates that the energy decrease is relatively higher than the increase in time for most power states.

3 QOS MONITORING AND SCHEDULING

Streaming applications process input data and produce output, iteration by iteration, and often have intuitive QoS requirements such as output per second (e.g., frames per second). Our proposed framework monitors the number of instructions and cycle count of each iteration, so that the iterations completing earlier than their deadlines can be slowed down to save power and energy. The cycle count is used to determine the accumulated slack and the current and past instruction counts are used to predict the number of instructions in future iterations. The prediction of the instruction count and the cycle count along with the slack are used to determine an appropriate core and frequency setting to execute the next iterations so that they finish close to the deadline in order to maximize power and energy savings.

The runtime monitoring approach is described in Section 3.1. Section 3.2 describes the instruction-count prediction approach. Finally, Section 3.3 completes the framework with an illustration of how scheduling decisions on a big versus LITTLE core is done based on the prediction and the available slack for heterogeneous multiprocessor platforms.

3.1 Runtime Monitoring and Time Slack

The applications that we consider typically consist of two parts: (1) an initialization and (2) a kernel. The initialization part prepares for kernel execution by, for example, reading input parameters and data. The execution time of this part is typically negligible in comparison with the total execution time. In contrast, the kernel, which is embedded in a loop, processes the input to produce the output and is the dominant part of the execution. Listing 1 illustrates an example, where the execution time of the initialization part is typically negligible compared to the kernel part, which executes for a large number of iterations.

The programmer must identify the most time-consuming loop and provide its QoS specification with a compiler directive as shown in Listing 1. The loop is then automatically instrumented with runtime library calls to read hardware-performance counters as shown in Listing 2. Each iteration is monitored to predict the computational requirement of future iterations and in conjunction with the QoS specifications to select a power state that minimizes the energy. In this context, X is the deadline per loop iteration and is specified in a compiler directive. Typically, the QoS specifications for streaming applications are provided in terms of number of frames per second and the calculation of the deadline per loop iteration (X) is straightforwardly derived from this.

Our framework proposes to use the instruction and cycle count from each loop iteration to estimate the computational demand for subsequent iterations and schedule them accordingly. A supervised execution of a loop, based on QoS specifications, can save considerable amounts of energy compared to a maximum-effort execution at full throttle. In short, our proposed framework

```
initialization();
#pragma QoS(X);
loop start
  kernel();
loop end
```

Listing 1. Loop QoS specification.

```
initialization();
loop start
  sloop_read_perf_counters();
  kernel();
  sloop_read_perf_counters();
  sloop_predict(instructions);
  sloop_schedule(prediction,QoS);
loop end
```

Listing 2. Loop instrumentation.

aims at executing every iteration on the lowest possible power state without missing the deadline to save energy.

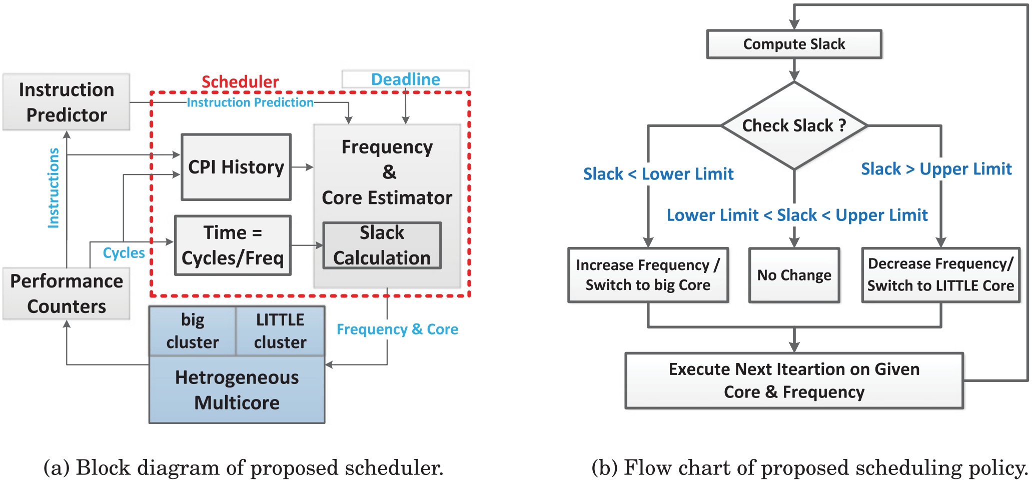
3.2 Instruction Count Prediction

The number of instructions in a given iteration of a loop depends on the control paths taken by the program based on input data. Such effects can be predicted by monitoring the trends in the number of executed instructions per iteration. We use the instruction count from the history of h previous loop iterations to predict the instruction count for the p future iterations, where h is the size of the history buffer and p is the prediction window. Initially, we simplistically assume that the predicted instruction count per iteration for the next p iterations is the same. Specifically, we refer to the history of instruction counts as $I[n]$, where n represents the iteration number. Furthermore, we refer to the number of predicted instructions for the next iteration as $I_P[n+1]$, where $P \in A, G$, (A) represents our proposed average predictor and (G) represents our proposed gradient predictor described in the following sections. We implement these predictors in a runtime system and analyze their performance and energy overheads in Section 5.

3.2.1 The Average Predictor. Averaging provides a reasonable mechanism to estimate future trends in a curve with relatively small overhead. The predicted number of instructions for the next iteration is expressed in Equation (1), where $I_A[n+1]$ is the prediction for the next iteration using the average predictor, $I[n-k]$ is the instruction counts from previous iterations, and k ranges from 0 to $h-1$, where h is the size of the history buffer. Despite its simplicity and low overhead, the average predictor adapts slowly to changes in the instruction count. Decreasing the size of the history window will certainly make the predictor response faster but will make it prone to noise, potentially making it oscillate. Therefore, we have to choose a conservative history window size, detailed in Section 4.

$$I_A[n+1] = \frac{1}{h} \sum_{k=0}^{h-1} I[n-k] \tag{1}$$

3.2.2 The Gradient Predictor. The rate of change between the number of instructions from consecutive iterations is a potentially more accurate mechanism to predict the behavior of an application. In this context, we apply geometric progression (GP) and use the current value as a base along with the rate of change from previous values to estimate a future value. A general expression for GP to estimate the kth term is given by the Equation (2), where r is the gradient (or rate of change), $a[n]$ is the current value, and $a[n+k]$ is the kth next value. The rate r is typically assumed to be constant; however, we dynamically revise it on every iteration from the instruction count of the last two iterations. When applications change phases, the gradient also changes and it is important to recalculate it during the entire course of execution. Similarly, the instruction count of the



(a) Block diagram of proposed scheduler. (b) Flow chart of proposed scheduling policy.

Fig. 4. Details of scheduling on a heterogeneous multiprocessor platform.

previous iteration is used as the new base. The prediction $I_G[n+1]$ for the next iteration is given by Equation (3).

$$a[n+k] = a[n] \times r^k \tag{2}$$

$$I_G[n+1] = I[n] \times \frac{I[n]}{I[n-1]} \tag{3}$$

However, the instruction count of an individual iteration may differ from the overall application trend and it can deteriorate the accuracy of a predictor based on Equation (3). We address this shortcoming by proposing a new predictor that uses Equation (5). We divide the history buffer into two equally sized groups and calculate the mean for each group. $Mean_1$ corresponds to instruction counts for iterations $n-h$ to $n-\frac{h}{2}$ and $Mean_2$ for iterations $n-\frac{h}{2}$ to n. The gradient is calculated as the ratio of $Mean_2$ over $Mean_1$ and is given by Equation (4). Moreover, $Mean_2$ is used as the new base to calculate the number of instructions for the next iterations.

$$r = \frac{Mean_2}{Mean_1} = \frac{\frac{2}{h}\sum_{k=0}^{\frac{h}{2}-1} I[n-k]}{\frac{2}{h}\sum_{k=\frac{h}{2}}^{h-1} I[n-k]} \tag{4}$$

$$I_G[n+1] = r \times \frac{2}{h} \times \sum_{k=0}^{\frac{h}{2}-1} I[n-k] \tag{5}$$

3.3 QoS-Based Scheduler for Heterogeneous Cores

The goal of the scheduler is to use the accumulated time slack and the prediction to allocate optimal resources to each iteration of the kernel, such that the execution completes close to the deadline. The execution starts at the highest frequency (e.g., in the context of our experimental system at 2GHz on the big core), and then future iterations are rescheduled on a suitable core and a frequency such that the accumulated time slack is minimized in order to save energy.

3.3.1 Operation. An overview of the operation of our proposed scheduler is depicted in Figure 4(a). The runtime scheduler is the primary controller that monitors each iteration of an application to predict future trends and to make scheduling decisions for future iterations. The sequence of operations performed by the scheduler is summarized as follows:

— The instruction count from the last iteration is injected into the predictor to get a prediction for the next iteration as discussed in Section 3.2.

— The predicted execution time is calculated based on the predicted instruction count. The scheduler maintains the history of CPI to calculate average CPI for both types of cores and uses these values to estimate the execution time. The details about execution time calculation are discussed in Section 3.3.2.
— The frequency and the selected core type for future iterations are estimated based on the predicted time, the time slack, and the deadline as described in Section 3.3.3.

3.3.2 Estimation of Execution Time. Our framework predicts the instruction count for future iterations and converts it to actual time in order to make scheduling decisions that respect the deadline. The execution time T for the next iteration is based on the predicted instruction count I and is given by Equation (6), where CPI is the estimated cycles per instruction and F is the frequency. We extend the model by deducting the timing overhead for SLOOP framework expressed in cycles, denoted as $SLOOP_{OH}$ in Equation (7). Please note that this value is the sum of the overheads for reading performance counters, prediction of instructions, and scheduling. These overheads are automatically added to the execution time, resulting in higher execution time, and thus it is important to compensate for the overheads in the estimation model of the new Frequency/Core; otherwise, it can cause missed deadlines.

$$T = \frac{I \times CPI}{F} \quad (6)$$

$$T = \frac{I \times CPI - SLOOP_{OH}}{F} \quad (7)$$

For the estimation of CPI we use the average CPI per type of core based on previously observed history. This approach provides a coarse but adequate estimate of CPI for a given application running on each core type. Workloads have different CPIs that mainly depend on the available instruction-level parallelism and memory-access pattern. So, the average CPI from recent history provides a coarse measure of these properties in real time. Since the execution always starts on a big core, we need an estimate of the LITTLE-core CPI until the first switch to a small core. A simple model of $CPI_{small} = CPI_{big} \times O$ is used for this scenario, where O is estimated offline and is discussed in detail in Section 4. After the first switch, we have CPI history for both types of cores and the offline estimate is no longer used. We extend Equation (6) to estimate the execution time at different frequencies on the same core and take into account the associated overheads by deriving Equation (8). The DVFS time overhead is denoted as T_{DVFS}. Similarly, to estimate the execution time on another type of core, we introduce the core switching overhead $T_{CoreSwitch}$ instead of DVFS overhead and derive Equation (9).

$$T = \frac{(I \times CPI) - SLOOP_{OH}}{F_{new}} + T_{DVFS} \quad (8)$$

$$T = \frac{(I \times CPI) - SLOOP_{OH}}{F_{new}} + T_{CoreSwitch} \quad (9)$$

3.3.3 Policy and Operating Modes. The prediction methodology proposed in this work is simple and effective. However, relying solely on a prediction to schedule future iterations may lead to either missed deadlines or faster-than-needed execution and higher energy consumption. Accurate predictions keep the accumulated slack low (ideally zero). On the contrary, less accurate predictions increase or decrease the accumulated slack beyond the expected margins. To this end, we augment our scheduler with a supervision mechanism that monitors the accumulated slack to assess the effectiveness of the predictor and safeguard against miss-predictions. An important effect of core switching is the cold misses that occur afterward. Our methodology automatically accounts for this in the following manner. The number of cycles required to execute the predicted number of instructions can increase after a core switch. This eventually results in reduction of the

slack. Since we measure the slack after every iteration, the effect of "cold misses after core switch" is taken into account by the scheduler, which uses the prediction and slack to schedule subsequent iterations.

We use a relatively simple policy that strives to keep the accumulated slack within a guard band and applies corrections to the prediction whenever the slack crosses the boundaries of this guard band. When the slack is within the allowable limits, the scheduler does not make new decisions.[1] A high-level flow diagram of the scheduling policy is illustrated in Figure 4(b).

The scheduler applies a correction C to the predicted execution time whenever the accumulated slack crosses the boundaries of the guard band. In such cases, a new core type and/or a new frequency are selected to push the slack back inside the guard band. If the slack increases beyond the upper limit, we slow down the processor (to consume slack) by adding the correction factor given by Equation (10) to the predicted time. GB_U and GB_L are the upper and lower limits of the guard band, respectively; p is the prediction window; and S is the accumulated slack. Here, the first term represents the difference between the slack and the upper level of the guard band and the second term is a constant that attempts to push the slack back to the guard band. Here, the scaling factor of 0.25 pushes slack only one-quarter into the guard band, while the factor p applies this correction to the whole prediction window. Similarly, if slack is below the lower limit, then we speed up the processor (to generate slack) by subtracting the correction factor shown in Equation (11) from the predicted time. Here, the first term represents the difference between the slack and the lower limit of the guard band and the second term is the constant described before.

$$C = S - GB_U + (GB_U - GB_L) \times p \times 0.25 \quad (10) \qquad C = GB_L - S + (GB_U - GB_L) \times p \times 0.25 \quad (11)$$

The selection of the guard-band boundaries is very important and allows for scheduling decisions that minimize the energy consumption without missing any deadlines. Since the guard-band selection is application dependent, we decide to set the guard-band boundaries in connection with the deadline set by the QoS specification. We set the guard band between $0.5 \times D_{QOS}$ and $1 \times D_{QOS}$, where D_{QOS} is the deadline per loop iteration derived by the QoS specification. This essentially means that the scheduler will try to keep the accumulated slack within 50% to 100% of the deadline per iteration. Moreover, if the scheduler does its job properly, then the full application will only complete no more than one D_{QOS} time before the last deadline. As will be discussed in Section 5, it is a big improvement over the unsupervised application execution.

4 EXPERIMENTAL METHODOLOGY

4.1 Hardware Platform and Configuration

We use an ODROID-XU3 board with the Samsung Exynos-5422 chipset (Hongsuk Chung 2013) for our evaluation. This chipset contains an ARM big.LITTLE architecture with a cluster of four Cortex A15 cores and a cluster of four Cortex A7 cores. The A15 is a performance-oriented, out-of-order core referred to as "big," while A7 is an energy-oriented, in-order core referred to as "LITTLE." Each core has a 32KB private L1 cache. The "big" cluster has a 2MB shared last-level cache (LLC), whereas the "LITTLE" cluster has a 512KB shared LLC. Energy consumption is measured using the four on-board sensors, each for the cluster of "big" and "LITTLE" cores, the GPU, and the DRAM. The measurements for each cluster include the four cores along with their private L1s and the LLC. For the evaluation, we consider single-threaded applications and use only one core in the cluster

[1] At the beginning of every new application, the predictors require some iterations to warm up and build their history; therefore we also do not make scheduling decisions during this warm-up phase.

Table 2. Workload Details and Baseline Energy Consumption

Suite	ALP										SPEC 2006		
Application	RayTrace							MPEG Encoder		MPEG Decoder	H264		
Input	teapot	balls	room	2balls	smallballs	sc98	msti	blah	teapot	blah	foreman base	foreman main	sss
Workload	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
Total Energy (J)	2,470	753	759	354	428	2,239	821	85	454	60.25	547	363	3,191
Energy/Iteration (J)	3.37	2.85	1.12	0.73	1.62	2.22	2.53	0.43	0.79	0.30	4.57	5.85	37

Table 3. Measured Statistics for the Average and Worst-Case Profiles

Workload	Average Instructions per Loop Iteration (Million)	Maximum Instructions per Loop Iteration (Million)	Minimum Instructions per Loop Iteration (Million)	Standard Deviation per Loop Iteration (Million)	Average Profile (Million)	Worst-Case Profile (Million)
W1	1,921	2,747	1,277	482	1,112	2,747
W2	1,379	1,749	889	298		
W3	652	818	455	126		
W4	431	495	351	52		
W5	813	993	551	155		
W6	1,124	1,579	760	203		
W7	1,466	1,841	1,026	283		
W8	556	685	189	101	510	685
W9	464	576	125	164		
W10	41	45	37	2	41	45
W11	4,055	4,285	417	657	12,162	34,737
W12	4,957	6,082	894	1,374		
W13	27,476	34,737	5,419	6,085		

at any given point in time. We also restrict the set of power states to the voltage-frequency pairs that have a voltage change along with frequency change. All unused cores are powered off except one LITTLE core, which executes the operating system thread.

4.2 Benchmarks

For the evaluation we use applications from the SPEC2006 (Henning 2006) and the ALP (Li et al. 2005) benchmark suites. In total, we experiment with 13 workloads including different input sets as shown in Table 2. Rows five and six show the total energy and average energy per iteration, respectively. The energy figures correspond to the baseline system used in our evaluation without employing our proposed scheduler.

4.3 Baseline System

We execute the selected set of benchmarks on the Cortex A15 big core at the highest frequency (i.e., 2GHz) and use this as the baseline. The total energy consumed during that execution is used as the reference and is shown in Table 2. The baseline models *race-to-idle*. It executes the workload at the fastest possible rate and then goes to sleep to save energy. Useful information is extracted from these runs for each application as shown in Table 3 and Table 1 and includes the maximum, minimum, and average instruction counts and execution time per loop iteration, respectively. The default deadline in each benchmark, denoted as D_{QOS} in the rest of the article, is the execution time of the slowest outer-loop iteration of the execution. The rationale is that different iterations have different execution times and in a real system the deadline is typically set such that even the slowest case completes before the deadline on a given computational platform. In order to cover a broad spectrum of use cases, we experiment with multiples of this base deadline in order to observe its effect on energy savings. In reality, programmers will define the deadlines based

on, for example, frame rates that can far exceed the execution time of the outer-loop iterations of benchmarks. Hence, setting the deadline to the slowest-running iteration results in a conservative estimate of energy savings.

4.4 Evaluated Systems

The proposed methodology is compared against a perfect prediction scheme and two offline schemes from related work (i.e., the average profile and the worst-case profile) (Suh et al. 2015). The perfect predictor is the one that can foresee the number of instructions to be executed in the next iteration and can allocate the optimal amount of resources. Since this cannot be done dynamically, we use a simulation-based methodology to establish the effect of a perfect predictor as follows. Each workload is executed on both the big and the LITTLE cores at each DVFS setting and the instruction and cycle counts and the energy consumed for each iteration are recorded in a trace, with all the nonactive cores powered off. These traces are then fed into a simulator that generates a schedule for the perfect predictor. The scheduler for the perfect predictor calculates the minimum required frequency and core type to meet the deadline for the next iteration using Equation (6). The simulated scheduler is fed with the instruction count of the next iteration to model the perfect prediction and the deadline is modeled as $D_{QoS} + 0.8 * S$, where S is the accumulated slack. We have found experimentally that if the whole slack S is added, there is a risk that the deadline is not met. By choosing 80% of the slack, the deadline is met.

The average and worst-case profiles here refer to the schemes in which the average and worst-case instruction count per loop iteration is used for scheduling. These values are also extracted from the baseline execution explained in Section 4.3. Average and worst-case profile values are calculated for each benchmark from the profile values of all the inputs to that benchmark and are provided in Table 3. The second and third columns represent the average and worst-case number of instructions per loop iteration for all the workloads. The sixth and seventh columns represent the average profile and worst-case profile value calculated for each benchmark. These instruction counts are fed to the core and frequency estimator as reference along with the average CPI and deadline to determine a schedule. Since the average and worst-case profile values calculated offline are constant, the schedule remains the same for the entire execution of the application. Please note that the slack-based scheduling proposed in Section 3.3.3 is not applied. It is important to note that the same inputs are used for both profiling and evaluation, making it the best case for the profile-based schemes.

4.5 Instrumentation

There are two important considerations for the experiments: (1) the instrumentation of loops and (2) a runtime system for scheduling. The first is performed by the programmer by specifying the "*#pragma QoS*(*Deadline*)." Based on this, the compiler inserts the required function calls at the appropriate points. However, in this study, we manually instrument the main loop in each benchmark with required runtime function calls. For the second, a user-level runtime system consisting of our proposed scheduler is implemented to schedule the iterations on the appropriate core and frequency. The "cpu-freq" library and the "sched_setaffinity" Linux system calls are used to switch between frequencies and the "big" and "LITTLE" cores at runtime.

The scheduling model provided in Section 3.3.2 requires the timing overhead for each component. We measure the overhead of each component and operation in isolation on our platform. Table 4 provides a summary of measured overheads for all the components and the operations. The components we want to quantify include monitoring, predictions, and scheduling. These components are typically executed in, on the order of, 1 microsecond on the platform. Unfortunately, the on-board energy sensors can only measure energy at a sampling rate greater than a millisecond.

Table 4. Timing and Energy Overheads of the Operational Units in SLOOP Framework

	LITTLE Core						big Core					
	Avg Pred	Grad Pred	Scheduler	Perf Monitoring	DVFS	Core Switch	Avg Pred	Grad Pred	Scheduler	Perf Monitoring	DVFS	Core Switch
Timing (cycles)	1,187	2,721	3,610	442	2-msec	6msec	756	2,115	2,515	470	2msec	6msec
Energy (nJ)	293	676	897	273	1,270μJ	3,810μJ	948	2,664	3,192	648	1,270μJ	3,810μJ

For this reason, we measure the energy for each of the components, by encapsulating them in a loop that executes a billion times and taking the average. Since we have to read the performance counters at the end of every iteration, we use a Linux kernel module that enables direct user-level access to the performance counters using the MRC instructions, making the overhead of reading performance counters minimal. The overhead for monitoring includes the overhead for reading the instructions, the cycles, and the energy consumed from the on-board energy sensor. The timing overheads for the frequency and the core switch are measured using the system timer over a large number of iterations in the baseline execution and then taking the average. The energy overhead is measured using the timing overhead and the power consumption. It is also important to point out that these overheads are negligible compared to the execution time and energy consumption of the iterations.

4.6 Experimental Parameters

Our framework uses CPI history to calculate the execution time from the number of predicted instructions, and since we always start execution on the big core, we need a CPI estimation for the LITTLE core to perform the first switch. Based on Shubham Kamdar (2015), we use the estimate $CPI_{LITTLE} = 2 * CPI_{big}$. This factor of "two" is only used to model the first switch to a small core. Afterward the CPI history maintained for each core type is used. Moreover, the prediction window size p is set to 20 for the scheduler and the history buffer size h for the predictor is set to 10.

5 EVALUATION

This section evaluates the gains provided by our proposed method and establishes the energy savings in comparison with the baseline case. First, the gradient predictor and the average predictor are compared against the average and worst-case offline techniques (profiles) using real instruction counts in Section 5.1 in order to measure prediction accuracy. Next, the performance of the scheduler is analyzed in Section 5.2, where the schedule using the average and worst-case profiles are compared against the schedule, based on the average and gradient predictors. A summary of the energy savings is presented in Section 5.3. Moreover, we also evaluate a perfect predictor with our scheduling scheme in order to establish the upper bound of energy savings. In the end, an analysis of overheads involved is presented in Section 5.4.

5.1 Instruction Prediction Accuracy

$$PredictionError(\%) = \frac{|\ (Prediction - Actual)\ |}{Actual} * 100 \quad (12)$$

Prediction error is a good metric to assess predictor accuracy and is calculated as the difference between the predicted and the actual number of instructions over the prediction window. The error is calculated over the actual values as shown in Equation (12). In this context, the standard deviation and the mean of percentage error are used for analysis. A summary of the results for each scheme is presented in Table 5, where the last row presents the geometric mean (GM) over all the applications. In general, the average and gradient predictors have a considerably

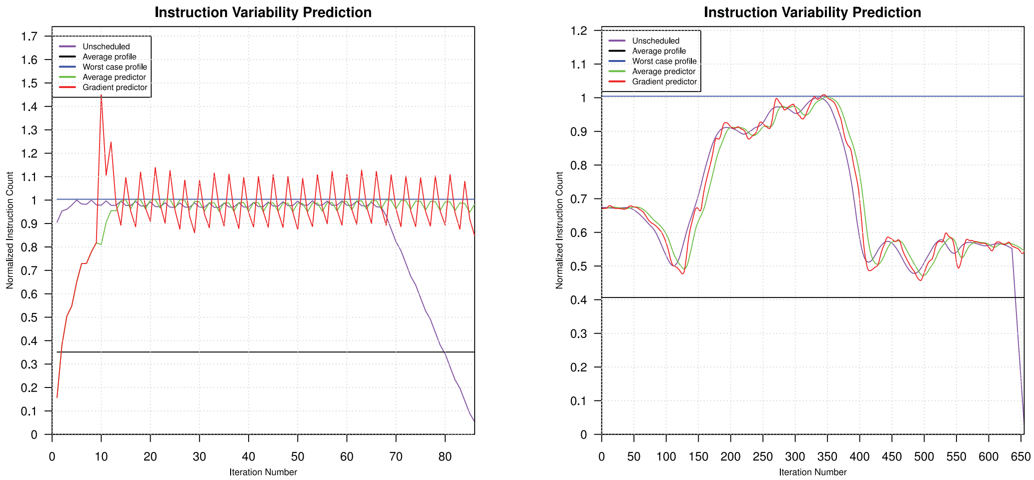
Table 5. Summary of Prediction Error for Gradient Predictor, Average Predictor, Average Profile, and Worst-Case Profiles

	Gradient Predictor		Average Predictor		Average Profile		WCET Profile	
Application	% Standard Deviation	% Average Error	% Standard Deviation	% Average Error	% Standard Deviation	% Average Error	% Standard Deviation	% Average Error
W1	3.8	5.5	5.4	7.1	13.6	41.6	32.8	44.5
W2	4.2	6.7	5.7	8.2	9.2	27.2	48.7	95.4
W3	2.7	5.1	3.6	5.8	36.0	70.3	88.8	320.5
W4	0.6	5.2	0.9	5.7	30.2	148.6	74.8	514.1
W5	4.3	6.2	5.7	7.7	28.6	32.7	70.9	238.1
W6	3.3	5.7	4.4	6.8	10.3	13.5	41.1	139
W7	2.7	5.1	3.6	5.8	15.7	24.5	39.5	86.9
W8	9.6	17.0	3	5.4	1.3	12.6	1.8	17.3
W9	7.4	13.8	2.6	5.0	3.9	5.5	6.4	40.7
W10	3.4	5.6	1.1	4.9	0.7	4.4	0.8	4.9
W11	13.9	8.3	7.7	6.7	0.9	173.7	2.5	681.9
W12	15.2	15.1	8.2	8.6	7.6	126.4	21.0	546.0
W13	9.1	11.6	3.8	5.0	0.3	58.4	1.0	18.5
GM	4.6	7.7	3.6	6.2	5.5	32.7	13.6	94.1

lower error rate compared to the offline average and worst-case profiles. Application behavior changes continuously over the course of the execution, and offline profiling-based estimations of instruction counts are by their nature incapable of adapting to changes. This is evident from the higher standard deviation and average error values for the average and worst-case profiles.

Our proposed predictors are capable of adapting to changes in the application behavior with considerable accuracy. The average prediction is stable but slow in reacting to changes in instruction counts per iteration. Consequently, it has lower accuracy for the RayTrace application (W1–W7). However, the gradient predictor adapts faster, compared to the average predictor, which is prone to high-frequency changes. For these reasons, the average predictor performs better than the gradient predictor for the applications that have largely constant behavior with small changes, that is, the MPEG decoder and encoder (W8–W10) and the H264 encoder (W11–W13). In contrast, the gradient predictor performs better for applications that have fast changing phases, that is, RayTrace (W1–W7).

Figure 5 presents a graphical view of predictor behavior for the selected applications, where the x-axis represents the iteration numbers and the y-axis represents the instructions normalized to the maximum of the actual number of instructions. The purple curve represents the actual number of instructions, where each point in the curve is the sum of the number of instructions for the next p iterations—the "prediction window." We use the ALP RayTrace workload (W1) and the SPEC2006 H264 workload (W13) for illustration. We observe that the gradient predictor and the average predictor react to the changes in the number of instructions per iteration. RayTrace has fast changing phases and the gradient predictor tracks the real instruction count with a small lag. However, the average predictor is slightly slower in following the application behavior. In the case of H264, the instruction count is mostly constant with small high-frequency changes. The average predictor performs better in this case and the gradient predictor oscillates trying to keep up with the changes. The average predictor is better for such applications because of its more stable nature. However, even in these situations, the prediction errors for both predictors are quite small and they perform far better than the offline techniques.



(a) SPEC2006 H264 input - sss

(b) ALP RayTrace input - teapot

Fig. 5. Predictor evaluation.

Table 6. Comparison of the Deadlines Missed for Different Schedules with the Deadline per Loop Iteration Set Equal to $1 * D_{QOS}$

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	AM
Grad Pred	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Avg Pred	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Avg Profile	97	48	0	0	0	74	98	38	0	0	0	0	84	34
WC Profile	0	0	0	0	0	0	0	0	0	0	0	0	0	0

In general, the average and gradient predictors are suitable for different scenarios. Both predictors achieve high accuracy with a mean average error of 4.6% and mean standard deviation of 7.7% for the gradient predictor and mean average error of 3.6% and the mean standard deviation of 6.2% for the average predictor across all applications. On the other hand, the average and worst-case profiles show much higher error statistics. In general, the online instruction prediction schemes perform considerably better than the profiling-based schemes.

5.2 Scheduling

The scheduler uses the prediction of the number of instructions and the accumulated slack to select the core type and the frequency to execute the future iterations of the loop so as to reduce energy consumption without missing any deadlines. Thus, one of the main metrics for our evaluation is the percentage of missed deadlines calculated over the total number of iterations. Table 6 presents our results. We present the arithmetic mean (AM) in the last column. First we note that the schedules based on the gradient and average predictors do not miss any deadlines. However, the schedules based on the offline average profile technique miss a large number of deadlines for several applications. The reason is that the average profile is calculated over all the inputs to a certain benchmark. This means that the resulting value is closer to the actual instruction count for one input than another. The schedules based on the worst-case profile, by construction, do not

miss any deadlines because they execute the application at full throttle and tend to overprovision the computational resources.

Our proposed scheduler tries to keep the slack within the guard band and makes decisions to change the core or the frequency only when the slack goes outside the guard band. An important observation is that the slowest iteration, whose execution time is used as the deadline, could potentially miss the deadline because of the overheads. However, this is not the case because the real deadline for each iteration is actually the sum of the absolute deadline plus the slack from the last iteration, that is, $D_{QOS} + S$. Since the guard band is set between $0.5 * D_{QOS}$ and $1 * D_{QOS}$, the last iteration will finish its execution at a time between 50% and 100% of its deadline. So the effective deadline is at least $1.5 * D_{QOS}$, thus offering considerable margin to tolerate the overheads generated from our framework.

Two inputs from RayTrace, teapot (W1) and sc98 (W6), are picked for graphical illustration of the slack, frequency, and core changes. Only the last hundred iterations of execution are shown in Figure 6. In the top four diagrams, the changes in the slack are shown, with the x-axis representing the iteration number and the y-axis representing the normalized slack with respect to the deadline per loop iteration (i.e., D_{QOS}). The top two diagrams show the behavior of slack for the average and the worst-case profiles for two inputs, where the dashed black line represents zero slack. The two middle diagrams show the behavior of the slack for the gradient and the average predictor. The two bottom-most diagrams represent the frequency and core schedule over the course of the execution, where the x-axis shows the iteration number and the y-axis represents the frequency and core settings. Letters "B" and "L" on the y-axis's label represent the big and LITTLE core, respectively, while the number represents the frequency in MHz.

The schedule based on the worst-case profile executes the application at full speed and thus overprovisions computational resources. As a result, the slack increases continuously throughout the execution of the application and ends up around 200 at the end of the execution. Since this value is normalized to D_{QOS}, it means that the application finishes 200 deadlines earlier than required. It also implies that there is a considerable margin for slowdown. It is important to note that the worst-case-profile-based schedule shows exactly the same behavior as unscheduled execution in this particular setting of the deadline. Another interesting fact worth noticing is that the worst-case profile's slack for W6 ends up at around 400. This means that the margin for slowdown varies a lot across the workloads and our methodology of prediction can capture this behavior at runtime and apply appropriate scheduling decisions.

The average-profile-based schedule also allocates a fixed amount of the resources (i.e., frequency and core) to the application execution. Since this value is based on the average across a set of inputs, it might be overprovisioning or underprovisioning resources. Obviously, it is less pessimistic than the worst-case profile, which is why the slack for the average profile for W1 decreases and that for W6 oscillates around the zero mark. This fact is also evident from Table 3, where the average instructions per loop iteration for W1 and W6 are 1,921 and 1,124 million, respectively, while the resulting profile value calculated from all the inputs to "RayTrace" is 1,112 million instructions per loop iteration. So the resulting average profile value is more suited to W6 than W1. This is the reason that the slack for W1 decreases and that for W6 remains close to zero. However, the negative values of slack imply that the deadlines are missed, and W1 misses more deadlines than W6 for average-profile-based scheduling, which is evident from Table 6, where W1 and W6 miss 97% and 74% of the deadlines, respectively.

In the schedules that are based on the average and gradient predictors, the frequency and core changes according to computational demands of application and slack are quite successfully managed inside the guard-band region. The gradient predictor being more accurate shows less and smaller overshoots outside the guard band compared to the average predictor. There are two

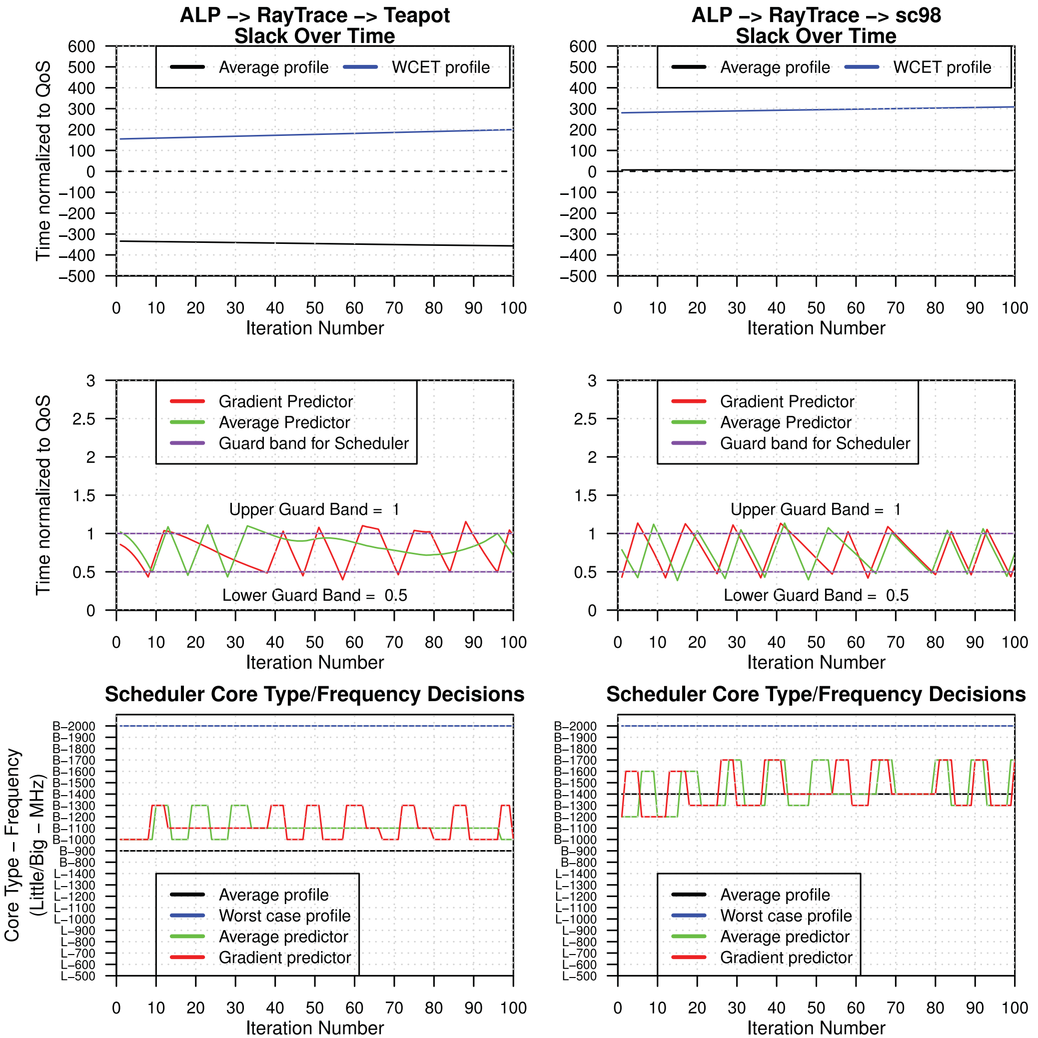


Fig. 6. Core and frequency schedule and its impact on slack for ALP RayTrace benchmark for input teapot (leftmost diagrams) and for the input sc98 (rightmost diagrams).

important factors that contribute to managing slack within the defined limits: (1) the prediction is constantly updated trying to keep up with the changes in instruction counts per loop iteration and (2) the scheduler makes a correction to the deadline whenever slack crosses the higher or lower limit of the guard band. If the prediction is always accurate, the slack should stay within the limits set by the guard band. However, since the application is changing phases, the prediction could suffer from the lag, underprediction, and overprediction that affects slack and push it outside of the guard band. In this context, the big changes in frequency and core are primarily driven by the phase changes in the application. However, small changes are the result of corrections that are applied by the scheduler whenever the slack crosses the boundaries of the guard band.

In summary, offline estimations of instruction counts (i.e., the average and worst-case profiles) are static estimations that are not capable of adapting to the changes in applications. Schedules

Table 7. Energy Improvement for Schedules Based on the Gradient and Average Predictors and the Average and Worst-Case Profiles with Deadlines Set to $1 * D_{QOS}$

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	AM
Perf Pred	37.47	33.79	32.35	29.11	32.18	42.17	34.35	36.28	35.59	37.63	16.52	29.86	26.34	32.59
Grad Pred	30.95	22.48	24.69	20.31	22.45	36.06	25.31	18.87	19.29	18.19	13.36	21.9	21.91	22.75
Avg Pred	31.37	23.24	24.93	19.72	21.12	36.05	25.47	27.68	29.95	18.09	9.91	28.26	26.64	24.80
Avg Profile	53.93	43.6	0	0	0	41.98	44.72	36.08	0	23.87	0	0	60.89	23.47
WC Profile	0	0	0	0	0	0	0	0	0	0	0	0	0	0

based on these estimates are not capable of managing slack within the allowable margins and cannot avoid missing deadlines, as the average profile-based schedule misses 34% of deadlines on average. On the other hand, our proposed online predictors (i.e., the average and gradient predictors) can adapt to the changes in application behavior, and the schedules based on these prediction schemes can successfully manage the slack without missing any deadlines.

5.3 Energy Consumption

$$EnergySavings(\%) = \frac{E_{base} - E_{new}}{E_{base}} * 100 \tag{13}$$

There exist two types of opportunities to reduce energy consumption. First, there is the inherent variation between the execution time across iterations, and this fact was elaborated on earlier using CDF of execution times per iteration in Figure 1. The marginal energy savings available in this case are studied by setting the deadline equal to $1 * D_{QOS}$. A summary of results for this case is presented in Table 7, where the percentage of energy savings for the perfect, gradient, and average predictors and the average and worst-case profiles are shown for different applications. The energy savings are calculated using Equation (13). Since the values can be zero or negative, arithmetic mean (AM) is used to present the average. This applies to all of the following tables showing energy savings. The schedule generated from the worst-case profile executes the application at full throttle and consequently there are no energy savings. Schedules based on average and gradient predictors show 24.68% and 22.94% energy savings, respectively. The upper bound of energy saving (i.e., 32.59%) is of course the one shown by the perfect predictor.

The average profile shows significant energy savings for some applications and no energy savings for other applications for the reasons discussed in the previous section. In some cases, the schedule based on the average profile has higher energy savings compared to the gradient and average predictors. However, in these cases, the average-profile-based schedule misses a large number of deadlines as shown earlier in Section 5.2 in Table 6. For example, in case of the W1, the schedules based on the average profile and the average and gradient predictors have energy savings of 54%, 31%, and 31%, respectively, for the deadline of $1 * D_{QOS}$. The energy saving for the schedule based on the average profile is certainly higher than schedules based on online gradient and average predictor. However, the average-profile-based schedule misses 97% of the deadlines, whereas the schedules based on gradient and average predictors do not miss any deadlines.

The second opportunity to save energy is created by relaxing the deadlines further to multiples of D_{QOS}. In this scenario, the margin for slowdown is increased for all the iterations whether they are slow or fast. We have analyzed our framework for three cases, where deadlines are set equal to $1.2 * D_{QOS}$, $1.5 * D_{QOS}$, and $2 * D_{QOS}$ and results are presented in Table 8. As expected, the energy savings are increased when increasing the deadline, but the improvement is not linear with respect to the increase in the deadline. The reason is that the voltage change associated with

Table 8. Energy Improvement for Schedules Based on the Gradient and Average Predictors and the Average and Worst-Case Profiles with Deadlines Set to $1.2 * D_{QOS}$, $1.5 * D_{QOS}$, and $2 * D_{QOS}$

	$1.2 * D_{QOS}$					$1.5 * D_{QOS}$					$2 * D_{QOS}$				
App	Grad Pred	Avg Pred	Avg Prof	WC Prof	Perfect Pred	Grad Pred	Avg Pred	Avg Prof	WC Prof	Perfect Pred	Grad Pred	Avg Pred	Avg Prof	WC Prof	Perfect Pred
W1	44.3	44.3	69.7	30.3	49.1	55.3	55.1	75.3	43.5	58.1	64.3	64.4	81.8	50.6	69.2
W2	41.6	41.9	48.8	0.0	45.4	50.6	50.8	52.8	0.0	55.1	62.1	63.4	74.5	34.6	64.2
W3	38.9	40.2	0.0	0.0	45.8	48.6	48.7	17.4	0.0	51.6	56.8	56.8	42.6	0.0	60.5
W4	38.4	38.4	0.0	0.0	44.1	47.4	47.8	0.0	0.0	51.3	56.9	57.1	0.0	0.0	58.7
W5	39.8	40.7	15.2	0.0	44.6	49.6	49.1	39.4	0.0	54.1	62.4	61.7	45.4	0.0	62.9
W6	46.8	46.9	47.3	0.0	50.4	57.1	57.1	53.3	0.0	59.6	68.2	68.3	71.7	26.8	72.1
W7	40.7	40.6	51.3	0.0	45.3	49.0	49.1	55.1	0.0	52.3	57.4	57.4	66.7	37.6	60.6
W8	40.2	42.1	45.5	26.2	49.9	49.7	50.8	50.7	43.7	56.2	60.8	65.1	55.8	50.7	63.6
W9	40.2	50.7	39.3	0.0	56.6	57.6	59.9	53.4	24.8	64.5	80.5	83.1	62.5	53.4	87.1
W10	38.1	47.7	51.3	37.0	57.9	61.5	61.1	60.4	54.8	69.2	71.5	72.1	70.3	66.7	77.0
W11	38.8	38.3	0.0	0.0	46.4	49.5	49.4	0.0	0.0	53.6	56.7	56.8	0.0	0.0	60.7
W12	33.0	40.3	0.0	0.0	46.2	42.4	45.9	0.0	0.0	54.1	51.6	52.0	0.0	0.0	61.0
W13	38.3	39.9	65.9	25.6	45.4	45.6	46.4	72.0	40.6	52.9	53.7	54.1	74.5	47.0	61.5
AM	40.0	42.5	33.4	9.2	48.2	51.0	51.6	40.8	16.0	56.4	61.8	62.5	49.7	28.3	66.1

Table 9. Energy Savings Using DVFS-Only on a Big Core with Our Proposed Predictors and Scheduler

QoS	Predictor	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	AM
1.2	Gradient	44.1	40.4	40.1	38.4	39	46.1	40.2	40.1	40.9	44.9	38.2	33.9	34.6	40.1
	Average	44.0	40.5	40.3	38.3	38.3	46.4	40.5	43.3	51.0	46.9	38.6	40.2	38.3	42.1
1.5	Gradient	51.4	48.9	48.5	48.1	47.7	53.1	48.7	49.1	58.2	62.0	49.5	44.2	45.5	50.4
	Average	51.4	49.1	48.5	47.6	47.8	53.1	48.7	50.1	59.9	62.3	49.5	45.3	45.8	50.7
2	Gradient	56.6	55.2	53.8	54.9	55.0	56.7	53.8	55.9	65.3	72.3	56.5	49.6	51.3	56.7
	Average	56.5	55.2	53.8	54.9	54.9	56.7	53.8	55.8	65.3	72.4	56.5	50.3	51.4	56.7

Table 10. Usage Statistics of LITTLE-Core (% of Iterations) with Deadlines Set to $1.2 * D_{QOS}$, $1.5 * D_{QOS}$, and $2 * D_{QOS}$

QoS	Predictor	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	AM
1.2	Gradient	3	17	0	0	16	6	0	0	0	0	0	2	0	3.4
	Average	2	17	0	0	16	7	0	0	0	0	1	0	0	3.3
1.5	Gradient	42	21	19	0	20	42	19	20	0	0	0	5	0	14.5
	Average	42	22	19	0	20	43	19	27	0	0	2	0	0	14.9
2	Gradient	65	52	45	33	57	83	46	56	86	18	7	2	33	45.0
	Average	64	62	42	33	52	83	44	70	98	8	6	2	27	45.0

each voltage-frequency step is not linear, so the slower we go, the less are the energy savings available.

An important factor in this scenario is the usage of the LITTLE core. We present the LITTLE core usage of our scheduler for various deadline specifications in Table 10. Moreover, in order to isolate the effects of core switch, we measure energy savings with a restricted version of our scheduler that uses only DVFS on the big core and present these results in Table 9. As the usage of LITTLE

Table 11. Final Slack Values for the Various Schedules for Deadline of $2 * D_{QOS}$

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	GM
Perf Pred	0.84	0.65	0.93	0.67	0.61	0.78	0.88	0.78	0.77	0.69	0.71	0.66	0.71	0.74
Grad Pred	0.8	1.03	0.89	0.72	0.47	0.7	0.43	1.03	0.96	1	0.51	1.13	0.89	0.78
Avg Pred	0.45	0.42	0.75	0.58	1	0.93	0.92	0.44	0.83	1.04	0.76	1.18	0.74	0.77
Grad Pred (DVFS)	90.7	8.3	67.0	0.99	0.49	138	67.9	5.9	45.4	0.96	1.03	4.41	3.04	8.23
Avg Pred (DVFS)	90.7	8.3	66.8	0.99	1.22	138	68.5	4.9	45.4	1	0.66	3.02	2.96	8.19
Avg Profile	−389	−47	300	275	85	21	−184	5.18	116	18	64	36	−87	N.A.
WC Profile	227	134	413	275	156	607	325	56	251	35	64	36	24	199

core increases, the energy difference between DVFS-only scheduling (Table 9) and heterogeneous core scheduling (Table 8) becomes higher.

We observe that if the slowdown margin is high, the DVFS-only scheme cannot efficiently exploit the available slack and turn it into energy savings. This slowdown margin is largely dependent on the deadline. Relaxing the deadline increases the usage of the LITTLE core and increases the effectiveness of a core switch.

Consider, especially, the average predictor for W9 when the deadline is $2 * D_{QOS}$. In this case, the LITTLE core usage is 98% and the energy saving, compared to the baseline (race-to-idle), is 83%, whereas the energy saving for DVFS-only is 65%. This means that with DVFS-only scheduling, the energy consumption is *2.8 times* smaller, while with our heterogeneous core scheduling scheme, the energy consumption is *5.8 times* smaller. We find that context switching to LITTLE cores provides an important avenue for energy savings.

Another statistical view of scheduling efficiency is to see how well the various schemes manage to complete the application execution closer to the deadline. Table 11 provides final slack values for different schemes. GM is used to calculate the average and is presented in last column. Here "N.A." is an abbreviation for "not applicable" because the GM can not be calculated for negative values. These values are normalized to the given deadline. That is, a value of "1" means that the application finished one deadline time before the final deadline. We see that the profiling-based schemes fail to exploit the slack and slow down the workload's execution appropriately. The gradient, average, and perfect predictor perform considerably well, with perfect predictor understandingly better than the former two. The gradient and average predictor schemes with DVFS-only perform mediocre and finish with a large amount of slack on hand. These experiments confirm our assumption that we can achieve higher energy savings when heterogeneous processing elements with varying power/performance operating points are available.

Our proposed framework can reduce power as well as energy. Because the input data-driven variations in execution time per loop iteration are significant, it is possible to achieve just-in-time completion and save energy. In case of a tight deadline of $1 * D_{QOS}$, it is essentially the slowdown of the faster iterations in order to complete close to the slower iterations that results in energy savings. In case of relaxed deadlines (i.e., $1.2 * D_{QOS}$, $1.5 * D_{QOS}$, $2 * D_{QOS}$), the execution times of both faster and slower iterations are slowed down to complete close to the new deadline and achieve further energy savings.

We finally study the impact of selecting the guard-band parameters on energy savings. Initially, the guard band is selected to range from $0.5 * D_{QOS}$ to $1 * D_{QOS}$. This selection means that the application completes in the second half between the two deadlines. Consequently, the final accumulated slack stays within this limit. This is a considerably pessimistic setting considering the initial proposal of finishing close to deadline and the energy savings also reduce significantly. The

Table 12. Energy Improvement for Schedules Based on Gradient Predictor and Average Predictor with Deadlines Set to $1 * D_{QOS}$ and Guard Band Set between $0.05 * D_{QOS}$ and $0.1 * D_{QOS}$

		W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	AM
Energy Savings	Grad Pred	33.4	28.6	27.4	24.3	25.9	38.6	29.3	27.1	24.7	28.3	22.1	19.0	27.2	27.4
	Improvement (%)	7.6	29.8	9.6	21.5	17.9	7.2	17.0	28.9	23.4	23.2	70.2	−5	4.5	19.7
	Avg Pred	33.0	27.9	27.3	23.7	25.0	38.1	28.9	35.3	35.7	23.3	19.2	26.2	31.0	28.8
	Improvement (%)	6.3	21.3	9.1	18.7	13.6	5.7	15.6	26.0	18.9	−6.7	92	−6.5	29.1	18.7
Deadlines Missed (%)	Grad Pred	0	0	0	0	0	4.75	0	34.1	9.0	0.5	0	0	6.9	4.3
	Avg Pred	5.9	8.7	0	0.05	4.1	5.5	0.5	0	0	0	0	0	0	1.9

rationale here is that this guard band is chosen over a set of applications and it represents a safe choice, where not many of the deadlines are missed. In fact, if the guard band is lowered and tightened, the energy savings are expected to increase, but the tradeoff is that additional deadlines can be missed. Table 12 shows such a scenario where the guard band is changed to $0.05 * D_{QOS}$ and $0.1 * D_{QOS}$ and the energy savings and missed deadlines are presented. This new guard band is set so that the application finishes between 5% and 10% of the time between two deadlines.

The energy savings are considerably improved for most of the cases. However, the number of missed deadlines increases as well. Considering the fact that different predictors are more accurate for different sets of applications, the increase in the number of missed deadlines is consistent with the prediction accuracy. The gradient predictor has low accuracy compared to the average predictor for the H264 encoder (W8–W9) and MPEG encoder (W11–W13). Consequently, the number of missed deadlines is higher for the gradient predictor than for the average predictor. Similarly, the average predictor has slightly lower accuracy compared to the gradient predictor for the RayTrace (W1–W7) and consequently the number of missed deadlines for the average predictor is more than the number of missed deadlines for the gradient predictor.

In summary, the setting of the guard band has an effect on energy savings. However, this selection is largely application dependent and has a slight effect on the number of deadlines missed. Changing the guard band from $0.5 * D_{QOS}$ and $1 * D_{QOS}$ to $0.05 * D_{QOS}$ and $0.1 * D_{QOS}$ increases the energy savings by 19.7% and 18.7% for the gradient- and average-predictor-based schedules, respectively. However, the number of missed deadlines is slightly increased by 4.3% and 1.9% for the gradient and average predictor, respectively.

5.4 Overhead Analysis

For our proposed method to work, the overheads incurred must be small. A summary of timing and energy overhead incurred by our proposed scheduling methodology is presented in Table 13. Here, we only provide the results for the gradient-predictor-based schedule as it is more computationally demanding and will have higher overheads. The overhead (shown in Table 13 as a percentage) is calculated as the ratio of overhead over the entire execution, where we show the ratio of timing and energy overhead to execution time and total energy, respectively, for each application. Overall, as we can see, the overhead is negligible for most applications. For some applications (e.g., W10), the timing overhead is somewhat higher (a few percent) but the energy overhead is still negligible. The maximum geometric mean of overhead, considering different QoS requirements, is only 0.062% and 0.076% for timing and energy overhead, respectively. Note that we have accounted for the energy overheads when reporting on the energy-saving gains earlier in Section 5.3.

Finally, we analyze the contributions to the overhead in detail in Table 14 and Table 15. Since the overheads are highest for the timing constraints of $2 * D_{QOS}$, we only present the detailed timing and energy overheads for this deadline. The timing and energy overheads are presented as

Table 13. Summary of Timing and Energy Overhead in Percent of the Gradient-Predictor-Based Schedule with Deadlines Set to $1 * D_{QOS}$, $1.2 * D_{QOS}$, $1.5 * D_{QOS}$, and $2 * D_{QOS}$

QoS	Overhead	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	GM
1	Timing	0.01	0.02	0.05	0.08	0.03	0.02	0.48	0.18	2.28	2.28	0.01	0.03	0.01	0.05
	Energy	0.006	0.009	0.019	0.026	0.013	0.013	0.009	0.154	0.059	0.089	0.003	0.008	0.002	0.0144
1.2	Timing	0.02	0.03	0.04	0.09	0.05	0.03	0.02	0.41	0.32	4.02	0.02	0.06	0.01	0.06
	Energy	0.011	0.016	0.027	0.046	0.031	0.022	0.011	0.218	0.165	0.207	0.007	0.025	0.003	0.028
1.5	Timing	0.022	0.016	0.071	0.073	0.044	0.038	0.031	0.805	0.278	2.064	0.015	0.065	0.005	0.062
	Energy	0.02	0.01	0.07	0.05	0.04	0.05	0.03	0.64	0.25	0.23	0.01	0.04	0.00	0.04
2	Timing	0.009	0.026	0.048	0.123	0.064	0.023	0.022	0.567	0.309	3.495	0.012	0.072	0.008	0.062
	Energy	0.015	0.039	0.070	0.144	0.098	0.050	0.033	0.766	0.794	0.641	0.012	0.065	0.009	0.076

Table 14. Detailed Timing Overheads of the SLOOP Framework Represented as Percentage to Execution Time for Deadline of $2 * D_{QOS}$

Overhead (%)	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	GM
Monitoring ($*10^{-4}$)	0.13	0.19	0.44	0.82	0.31	0.19	0.19	1.26	0.91	21	0.17	0.11	0.02	0.34
Grad Pred ($*10^{-4}$)	0.65	0.93	2.05	3.71	1.53	1.0	0.91	6.13	4.86	90.8	0.74	0.49	0.07	1.6
Scheduling ($*10^{-4}$)	0.2	0.51	0.72	1.32	0.89	0.35	0.32	4.03	4.11	39.6	0.49	0.43	0.06	0.82
DVFS ($*10^{-2}$)	0.69	0.99	2.62	5.46	2.21	1.22	1.05	22.0	19.89	200	0.89	2.88	0.35	3.07
Core-Switch ($*10^{-2}$)	0.17	1.57	2.14	6.81	4.15	1.05	1.16	34.29	10.95	144	33	4.32	0.45	2.81
Total ($*10^{-2}$)	0.87	2.57	4.79	12.33	6.39	2.28	2.22	56.69	30.94	349	1.23	7.21	0.84	6.19

Table 15. Detailed Energy Overheads of the SLOOP Framework Represented as Percentage to Total Energy for Deadline of $2 * D_{QOS}$

Overhead (%)	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	GM
Monitoring ($*10^{-4}$)	0.48	0.59	1.34	2.05	1.05	0.92	0.6	3.89	4.66	7.37	0.33	0.23	0.04	0.88
Grad Pred ($*10^{-4}$)	1.03	1.53	3.74	6.39	2.48	1.43	1.64	9.57	6.9	27.89	1.27	0.72	0.11	2.26
Scheduling ($*10^{-4}$)	0.28	0.47	1.4	1.97	0.96	0.44	0.63	5.13	4.9	10.89	0.74	0.5	0.07	0.97
DVFS ($*10^{-2}$)	1.22	1.49	3.84	6.37	3.39	2.68	1.56	29.71	51.11	36.59	0.86	2.6	0.36	3.74
Core-Switch ($*10^{-2}$)	0.30	2.37	3.14	7.94	6.38	2.3	1.71	46.3	28.15	26.34	0.32	3.9	0.46	3.43
Total ($*10^{-2}$)	1.55	3.88	7.04	14.42	9.81	5.01	3.3	76.58	79.42	64.12	1.20	6.51	0.86	7.55

a percentage of the execution time and total energy, respectively. Here, we show the overheads for the monitoring, prediction, scheduling, DVFS, and core switching assuming the gradient predictor. Overall, the overhead (energy or timing) for monitoring, prediction, and scheduling is negligible, although the overhead for DVFS and core switching is somewhat higher. While monitoring and prediction happen in every iteration, the scheduling and DVFS (or core switching) overhead is only incurred whenever the slack is outside the allocated guard band. As a result, the overhead of them is less critical. The bottom line is that since the overhead of monitoring, prediction, and scheduling is negligible, overall the overhead is low. It should also be noted that since we use operating system calls to accomplish DVFS and core switching, there is considerable room for improvement in reducing these overheads.

6 RELATED WORK

QoS specifications have been attempted in the past to reduce energy consumption in computing systems. A methodology is presented by Hughes et al. (2001a), where the variability in execution

times of video encoders/decoders is studied and later used to propose a framework for prediction of the remaining execution time (Hughes et al. 2001b), as we do. However, their approach builds into the prediction strategy what type of frame is currently analyzed (I, P, or B). Hence, it is not application agnostic. In addition, they base their predictions on the longest executing frame in the recent past, which may yield pessimistic predictions and less energy savings compared to our approach. Kluge et al. (2010) also propose an execution-time prediction methodology based on autocorrelation clustering targeting video encoders/decoders. Similar to Hughes et al. (2001b), their approach works well because of the strong correlation between execution time and frame type (I, P, or B). In stark contrast to both of these works, our approach is general and application agnostic and can work robustly without any knowledge of the application.

Själander et al. (2012) present an interesting approach to energy savings in cellular base stations. The activity in base stations is measured over a time window and used as a prediction to allocate the computational resources (i.e., the number of cores) in the next time window. Idle cores are power-gated to save static power as well. Their approach is to always assign two extra cores than predicted as a safeguard against underprovisioning of system resources. They demonstrate considerable energy savings over normal execution with no resource management. Similar to the works already discussed, this work relies on application-domain knowledge.

Mohapatra et al. (2007) use QoS specifications to apply energy-saving adaptations at all layers of the system (i.e., applications, OS, network and processors). They define a number of QoS settings in terms of screen resolution, frames per second, and network speed that are developed based on user perception. The proposed framework assigns the highest possible performance point to keep the energy consumption below the given budget. Their QoS specification is based on user perception rather than on fixed deadlines. In contrast, our approach identifies and exploits the variability in execution time of kernels without sacrificing the quality (e.g., frame rate and resolution) by focusing specifically on processor performance. In addition, their approach targets video encoder/decoder applications, whereas our approach is general.

Holmbacka (2015) proposed a runtime manager for energy reduction based on maintaining the required performance. Again, the maintenance of the required performance level does not guarantee the maximum energy savings because applications' performance requirements change over time. Our approach to use QoS deadlines coupled with online application monitoring and progress tracking provides a better mechanism that saves a significant amount of energy.

Linux OS governors typically scale the performance (i.e., frequency core) based on utilization. A higher utilization simply results in a switch to higher frequency or a big core. Here it is important to note that the OS lacks the information about the user program's deadline and utilization is not an appropriate measure to determine the computational needs of programs. A scheme relying on this fact was proposed by Kim et al. (2014), where a new arriving task is assigned to a processor that is predicted to have the lowest utilization (percentage of activity) instead of the load (number of tasks in the processor queue) typically used by the Linux Completely-Fair-Scheduler (CFS). This results in keeping the utilization low and thus the governor keeps the processor in low-performance state. However, all these schedulers are agnostic of QoS deadlines and cannot make informed decisions. In contrast, our scheme provides the application with an interface to request QoS deadlines and our runtime ensures that deadlines are met with significantly lower energy consumption.

Suh et al. (2015) propose to provide determinism to real-time applications against the variability in the execution throughput of an out-of-order processor by stabilizing the MIPS rate using a control system. This work uses profiling to determine the target MIPS rate based on the worst or average case. A PID-based control system then tries to regulate the MIPS rate against target MIPS, thus completing applications before the deadline and saving energy. However, a worst-case value will be higher and the application may complete a lot earlier than the actual deadline. Similarly,

an average profile can have significant variability compared to a real profile and might suit one application more than another as we have shown in our evaluation. Instead, we use online information from hardware performance counters to continuously predict the execution rate. Moreover, different phases of the application require different execution speeds. In the case of streaming applications, which require their frames to meet deadlines, the average profile may lead to missed deadlines. Therefore, it is important to continuously monitor the application and adjust the execution rate.

7 CONCLUSIONS

This article proposes a general, application-agnostic, online prediction methodology for the number of instructions to be executed in the next phase coupled with a supervisory mechanism based on slack to achieve just-in-time completion of each computational phase. The proposed methodology instruments the outer loops of the application (which often constitute the bulk of the execution) and monitors the characteristics of a loop inside a runtime layer (the number of instructions and cycles per iteration) via existing hardware performance counters. The runtime layer predicts the number of instructions and the time for the whole execution of the program (or a part of it) based on the history and performs supervised loop execution (SLOOP) by controlling DVFS and selecting a core type (big O-o-O vs. LITTLE in-order core) that meets the QoS deadline.

This article makes a number of contributions: First, it proposes a lightweight runtime progress-tracking methodology based on the execution of the outer loops in the applications. Second, it contributes with a simple, yet accurate, instruction and execution-time predictor based on instruction-count history. Finally, it contributes with a novel scheduler for heterogeneous multicores that saves energy by tuning the frequency and the core type based on QoS, the prediction, and the available slack. It is experimentally shown that the overall proposed methodology saves at least 23% energy with a deadline that corresponds to the slowest iteration and 40% or more when the deadline corresponds to 20% longer time than the slowest iteration. We also show that the extra energy needed for the SLOOP framework is negligible.

REFERENCES

K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA'12)*. 213–224.

J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

H.-D. Cho, H. Chung, and M. Kang. 2013. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLETM Technology. Retrieved from https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf.

C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. 2001a. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, 254–265.

C. J. Hughes, J. Srinivasan, and S. V. Adve. 2001b. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*. 250–261.

S. Kaxiras and M. Martonosi. 2008. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers.

M. Kim, K. Kim, J. R. Geraci, and S. Hong. 2014. Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE'14)*. European Design and Automation Association Belgium, Article 223, 4 pages.

F. Kluge, S. Uhrig, J. Mische, B. Satzger, and T. Ungerer. 2010. Dynamic workload prediction for soft real-time applications. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*. 1841–1848.

R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*. 81–92.

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

S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. 2007. DYNAMO: A cross-layer framework for end-to-end qos and energy optimization in mobile handheld devices. *IEEE J. Sel. Areas Commun.* 25, 4 (2007), 722–737.

A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. 2012. Computational sprinting. In *Proceedings of the IEEE International Symposium on High-Performance Comp Architecture.* 1–12.

N. Kamdar and S. Kamdar. 2015. big.LITTLE architecture: Heterogeneous multicore processing. *International Journal of Computer Applications* 119, 1 (2015), 35–38.

J. Lilius, S. Holmbacka, and S. Lafond. 2015. Performance monitor based power management for big.LITTLE platforms. In *Proceedings of the HiPEAC Workshop on Energy Efficiency with Heterogeneous Computing.*

M. Själander, M. Martonosi, and S. Kaxiras. 2014. *Power-Efficient Computer Architectures: Recent Advances.* Morgan and Claypool Publishers. DOI: http://dx.doi.org/10.2200/S00611ED1V01Y201411CAC030

M. Själander, S. A. McKee, P. Brauer, D. Engdal, and A. Vajda. 2012. An LTE uplink receiver PHY benchmark and subframe-based power management. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems Software.* 25–34.

B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang. 2014. PPEP: Online performance, power, and energy prediction framework and DVFS space exploration. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.* 445–457.

J. Suh, C.-T. Huang, and M. Dubois. 2015. Dynamic MIPS rate stabilization for complex processors. *ACM Trans. Archit. Code Optim.* 12, 1, Article 4 (April 2015), 25 pages.

Received February 2017; revised September 2017; accepted September 2017