Performance-Energy Trade-off in CMPs with Per-Core DVFS

Solomon Abera(✉), M. Balakrishnan, and Anshul Kumar

Indian Institute of Technology Delhi, New Delhi, India
{solomon,mbala,anshul}@cse.iitd.ac.in

Abstract. In recent years, energy consumption of multicores has been a critical research agenda as chip multiprocessors (CMPs) have emerged as the leading architectural choice of computing systems. Unlike the uniprocessor environment, the energy consumption of an application running on a CMP depends not only on the characteristics of the application but also the behavior of its co-runners (applications running on other cores). In this paper, we model the energy-performance trade-off using machine learning. We use the model to sacrifice a certain user-specified percentage of the maximum achievable performance of an application to save energy. The input to the model is the isolated memory behavior of the application and each of its co-runners, as well as the performance constraint. The output of the model is the minimum core frequency at which the application should run to guarantee the given performance constraint in the influence of the co-runners. We show that, in a quad-core processor, we can save up to 51% core energy by allowing 16% degradation of performance.

Keywords: CMP · Shared resource · DVFS · Machine learning

1 Introduction

Over the last couple of decades, CMPs have been the leading architectural choice for computing systems ranging from high-end servers to battery-operated devices. Energy efficiency has been an issue for multicores due to battery life in portable devices, and cooling and energy costs in server class systems and compute clusters. Despite the fact that CMPs improve performance through concurrency, the contention for shared resources makes their performance and energy consumption unpredictable and inefficient [7,8]. These depend greatly on the nature of the co-runners.

Dynamic voltage and frequency scaling (DVFS) is used to reduce the power consumption of a processor by trading-off performance. In recent years, modern processors (Intel Haswell, IBM Power8, ...) provide support for per-core DVFS where each core can run at different frequency, resulting in a vast configuration space for the applications running on these cores.

M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 225–238, 2018.
https://doi.org/10.1007/978-3-319-77610-1_17

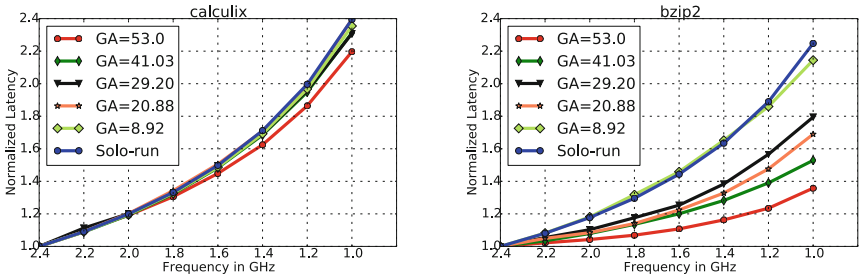
Compute-bound applications, which make very few accesses to LLC, benefit from a higher core frequency as their performance is determined by the processing speed of the cores.

On the other hand, memory-bound applications, which make a lot of accesses to the LLC, behave differently and can be divided into two classes. The first class consists of those applications whose performance has a high dependence on the shared cache space (applications with high data reuse, or "cache-friendly" applications). These show higher performance when they run alone or with compute-bound applications. With such co-runners, their performance is determined by the core frequency as their memory transaction latency is hidden by the cache. However, in the wake of competition for shared cache space from other memory-bound co-runners, their performance hugely drops. In such situations, the core frequency is not a big factor, and it can be lowered without impacting the performance much.

The second class of memory-bound applications are those whose performance does not depend on the amount of shared cache space (applications with low data reuse), like streaming applications. However, the performance of such applications is affected by the available memory bandwidth when they run with memory-bound co-runners. Regarding core frequency, varying it has little impact on the performance of these applications, regardless of the nature of the co-runners.

Any DVFS policy should take these determining factors into account before choosing the optimal frequency at which any workload should run. Consider the two SPEC2006 benchmarks `calculix` and `bzip2`. `Calculix` is a compute-intensive benchmark, whereas `bzip2` is a cache-friendly one. We simulated the execution of each of the two benchmarks running on a quad-core CMP sharing 2 MB L2 cache with other three co-runner benchmarks. We prepared five different sets of co-runners, each posing a different cumulative pressure on the shared L2 cache. We quantify the pressure posed by an application with the metric "aggressiveness" (see Sect. 3.1). The cumulative pressure of the three co-runners is termed "global-aggressiveness" (GA). The higher the GA, the greater the pressure on the L2 by the co-runners. When `calculix` runs with different competing benchmarks (Fig. 1a), its performance shows little degradation. Rather its performance is severely affected by the reduction of its core frequency. On the other hand, `bzip2` (Fig. 1b) shows different levels of performance degradation with different co-runners. When it runs with memory-intensive co-runners ($GA = 53$), its execution time increased by only 40% when the core frequency changed from 2.4 GHz to 1.0 GHz. However, when it runs with compute-intensive workloads ($GA = 8.92$), it slowed down by 120% for the same change in frequency. Therefore, any proposed model to select the appropriate frequency should take into account the application's characteristics and the global stress on the shared resources.

Furthermore, there are cases in which we may need a DVFS policy that enables us to trade-off certain percentage of the maximum achievable performance for energy savings. For example, let us say the user is willing to sacrifice



(a) calculix (b) bzip2

Fig. 1. Effect of DVFS and resource contention on performance

10% of the maximum possible performance of an application (when it runs with a given set of co-runners) in exchange for energy savings. As we discussed earlier, this 10% of application's performance is determined by its behavior as well as the nature of the co-runners. When that application runs with memory-intensive workloads, surrendering 10% of its performance might allow significant lowering of the core frequency as the application will slow down due to the cache and bandwidth contention. On the other hand, the same application, when it runs with less memory hungry applications, even a small reduction in the core frequency may reduce the performance beyond the allowed 10% limit. In this paper, we model the performance-energy trade-off using a learning-based algorithm. In order to capture the contention among the co-runners, we chose a lightweight contention metric that can efficiently convey the potential contention that will be faced by the workload. The model takes individual LLC aggressiveness, the global LLC intensity that emanates from the other cores and the performance constraint as input and generates the optimal frequency setting for that core. We assume that the underlying architecture to be a CMP architecture consisting of multiple symmetric cores, where all the cores share the LLC. We also consider the applications to be single threaded, with no data sharing among them.

The rest of the paper is organized as follows: Sect. 2 describes previous work in the domain of CMP energy efficiency. Section 3 discusses the overview of the model construction, Sect. 4 compares different machine learning algorithms, Sect. 5 describes the evaluation of the described scheme, for different performance constraints, and shows the efficiency of the developed model. Finally, Sect. 6 concludes the paper.

2 Related Work

Energy consumption has become an important optimization metric for CMP based computational platforms. Since its proposal by Weiser et al. [1], DVFS has been used to minimize the processor energy consumption while limiting the reduction of the overall system performance. DVFS can be applied in CMPs on a per-chip, per-core or per-cluster basis. Most of the previous works are directed

towards CMPs with chip-wise DVFS. There have been a significant amount of DVFS based works [2–6] focused on real-time systems that try to reduce energy consumption by utilizing the slack time for frequency scaling.

A lot of solutions have been proposed in the form of energy-aware scheduling that try to minimize the effect of contention, and apply DVFS to decrease the energy consumption. Merkel et al. [9] used task activity vectors (L2 and memory accesses) to capture the resource utilization of each task. When they schedule tasks, they try to pair memory-intensive tasks with compute-intensive ones to improve performance. When there are only memory intensive tasks in the workload, they scale down the chip frequency to save energy. Dhiman et al. [10] proposed a learning-based algorithm for a multi-tasking environment that suggests the optimal frequency based on tasks' degree of memory-boundedness. They used CPI stacks to quantify this behavior.

In recent years, machine learning algorithms have been applied to perform intelligent DVFS based energy saving [10–15]. The authors of [14] used reinforcement learning in which they took task characteristics and processor configuration to scale frequency for real-time systems. The task execution characteristics are derived from the execution time of the task (its CPU-time and stall-time). The proposal by Shen and Qiu [15] is the most related work to ours. In their work, they applied a machine learning technique to predict the performance degradation that would be faced by an application due to other applications in a CMP, and simultaneous application of DVFS. They define degradation with respect to a solitary run (solo-run) of the application on the CMP at the highest allowed frequency. They assumed global DVFS for all the cores. In this work, we assume that DVFS can be individually applied to each core. In addition to that, we argue that it would be difficult to guarantee a quality of service from the solo-run performance perspective as resource contention depends on the identity of the co-runner. Instead, the reference should be the maximum achievable performance with the given set of co-runners. In this work, we take this approach. For a given application, based on its memory behavior and that of its co-runners', the model can predict the correspondence between the performance loss and the discrete frequency steps.

3 Model Construction Methodology

The proposed machine learning based DVFS model attempts to predict the optimal core-frequency setting for a given user-specified acceptable loss in performance. The model is constructed offline by capturing the relationship between the nature of a benchmark, its operating frequency, the nature of its co-runners, and its performance. The model is then used online to find the optimal frequency that an application should run at, such that the given performance requirement is satisfied, and maximum energy savings are obtained.

Section 3.1 discusses the contention metrics that best capture the nature of a benchmark for the task at hand while Sect. 3.2 describes the data collection methodology. Section 3.3 then discusses the building process of the model and Sect. 3.4 covers the application of the model.

3.1 Contention Metrics

In order to construct a model that accurately captures the impact of frequency scaling and resource contention on performance, we need appropriate contention metrics. When applications are run in an isolated environment (solo-run), the amount of energy saved per percentage point of performance degradation varies based on their characteristics. Applications that are compute-intensive tend to save less energy for each percentage point of performance loss. Whereas, applications that are memory intensive tend to save much more energy for each percentage point of performance loss. This is because these applications take much time to progress as they wait for their data to arrive. Hence, in an isolated environment, the applications performance-energy trade-off balance can be modeled by its characteristics only.

As we mentioned earlier, shared resource contention in CMPs, particularly competition for shared cache space and memory bandwidth, severely harms performance and makes them energy inefficient. The slowdown encountered by individual applications hugely varies with the identities of the co-runners. The more the application slowed, the more insensitive it will be to frequency change and vice versa. Hence, the effect of frequency scaling also varies with the co-runners.

Therefore, when an application runs with other co-runners, they also prove to be a factor in the obtained energy savings through frequency scaling. If the application's performance does not depend on its usage of the shared resources, it does not incur any significant additional delay because of sharing. Hence, its own memory characteristics can be enough to drive the performance-energy trade-off and shows similar trade-off curve as its solo-run (Fig. 1a). If the application's performance does depend on the usage of the shared resources, it can show different behaviors based on the co-runners' characteristics. When the co-runners are not very shared resource hungry, the application might not incur much slowdown. Here, the performance-energy trade-off might not deviate much from its solo-run trade-off curve (see the curve in Fig. 1b with $GA = 8.92$). On the other hand, if its co-runners are resource hungry, the magnitude of their resource usage determines the slowdown suffered by the application. Accordingly, its energy saving per percentage point of performance loss varies with co-runners (see the curves in Fig. 1b with $GA = 53, GA = 41.03, GA = 29.20$ *and* $GA = 20.88$).

We desire a lightweight contention metric that conveys the resource hungriness of the application and helps in predicting slowdown that will be encountered by individual applications when they run together. This metric should be easy to collect online and the number of attributes should also be small as it reduces the sampling time. We have collected various performance metrics and tested their ability to predict the potential contention between the co-running applications. In our study, we run 275 combinations (each set of four) of SPEC2006 benchmark fragments on a quad-core environment sharing the LLC. We record the solo-run and co-run performance of the four co-running applications A_0, A_1, A_2 and A_3. The solo-run and co-run performance of each application is represented

by its number of instructions per cycle as IPC_{solo} and IPC_{co} respectively. We computed the average slowdown as specified in Eq. 1.

$$Slowdown = \frac{\sum_{i=0}^{3} IPC_{solo}(A_i) - \sum_{i=0}^{3} IPC_{co}(A_i)}{\sum_{i=0}^{3} IPC_{solo}(A_i)} \times 100, \quad (1)$$

We have analyzed parameters that present a high correlation with the observed slowdown. We chose three metrics: the solo-run `IPC`, number of LLC accesses (`LLCA`) and LLC misses per 1K cycles (`MPKC`). We also considered two methods of aggregating the parameters of the applications: *sum* and *product*. Table 1 presents the correlation between the six different candidates with the actual slowdown.

Table 1. Correlation between aggressiveness strategies and slowdown

Parameter	Correlation with slowdown
IPC_{sum}	−0.69469
$IPC_{product}$	−0.46521
$LLCA_{sum}$	0.73869
$LLCA_{product}$	0.48090
$MPKC_{sum}$	0.81011
$MPKC_{product}$	0.57714

As we can see from Table 1, $\texttt{MPKC}_{sum}$ shows the highest correlation with slowdown. The `IPC` based contention metrics show negative correlation with the slowdown, as high aggregate `IPC` implies lower contention, resulting in a lower slowdown. In this work, we represent the memory characteristics of applications by their aggressiveness scores ($A\_\texttt{Score}$). $A\_\texttt{Score}$ is a metric that characterizes how aggressively an application competes for the shared cache space and memory bandwidth. We use `MPKC` as $A\_\texttt{Score}$, and to model the global cache pressure emanating from the co-runners, define global-aggressiveness (GA) as the sum of the individual $A\_\texttt{Scores}$ of the co-runners.

We also perform experiment to demonstrate the correlation of the $A\_\texttt{Score}$ with performance loss, under the influence of frequency scaling. We run 20 single-threaded, single-phase (collected 250 million instructions fragments using SimPoint) SPEC2006 benchmarks on sniper multicore simulator with quad-core configuration sharing a 2 MB L2 cache. In the experiment, we run the benchmarks without competing co-runners. We scaled the frequency from 2.4 GHz through 1.0 GHz and record the performance response of each benchmark to the frequency scaling. We present, in Fig. 2, the correlation between their $A\_\texttt{Score}$ and performance degradation. As we can observe from the figure, the compute-intensive benchmarks (low $A\_\texttt{Score}$) show higher performance degradation when their frequency is scaled down. On the other hand, we can see memory-intensive

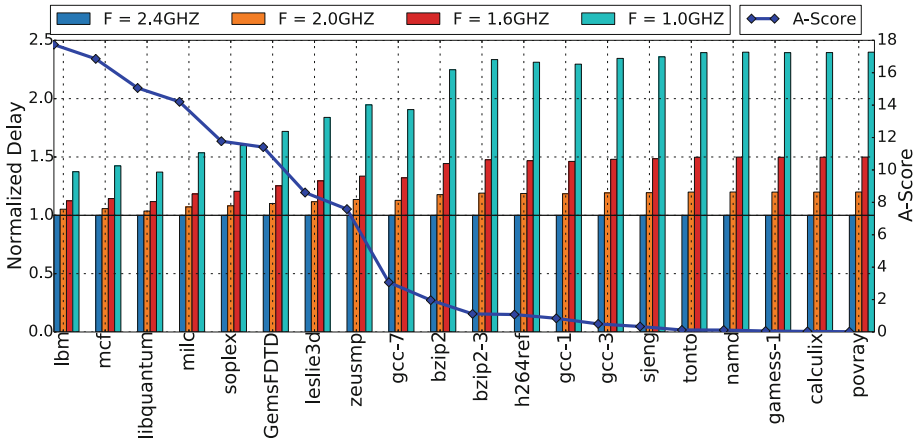


Fig. 2. Sensitivity of 20 SPEC2006 applications for frequency scaling and correlation with their *A*_Score

benchmarks are only marginally affected by frequency scaling as their performance is not determined by the core frequency.

In addition to its correlation with the slowdown (because of contention as well as frequency scaling), *A*_Score also can capture the impact of core-frequency changes on the aggressiveness of the application. Let us assume one application has x LLC misses when running at frequency f1. When the same application runs at f2 (f2 = 0.5f1), its LLC misses do not change but its cache pressure is minimized as its cache requests arrive at longer time intervals between them. This phenomenon is captured by the choice of LLC MPKC as a contention metric and makes it suitable for online DVFS modeling as every core might run at a different frequency. The cycles in our context are representation of time (elapsed time multiplied by the maximum core frequency).

3.2 Data Collection

Let us assume a processor having N_C cores. Let us also assume we have a set of N_A applications $A = \{A_0, A_1, \ldots, A_{N_A-1}\}$ that are to be run on this processor. We first collect the aggressiveness score of each application $A_{cur} \in A$ as $A\_Score_{A_{cur}}$ by running (or simulating) it alone on the given processor. We then construct a set of N_{CG} co-runner groups, $C = \{CG_0, CG_1, \ldots, CG_{N_{CG}-1}\}$, with each group having $N_C - 1$ applications from the set A. The global-aggressiveness of each co-runner group C_{cur} is given by $GA_{C_{cur}}$, and is computed by summing the individual *A*_Scores of the $N_C - 1$ applications in that group. We prepared the set C in such a way that their GA values are well spread over the entire spectrum of GA values ranging from maximum (all memory-bound) to the minimum (all compute-bound). We take only representative samples, not exhaustively, to construct the set C.

Algorithm 1. Data Collection Methodology

```
1  for each application A_cur in A do
2      schedule A_cur on core 0;
3      for each co-runner group C_cur in C do
4          schedule the applications in C_cur on cores 1 to N_C − 1;
5          set the frequency of cores 1 to N_C − 1 to f_max;
6          set the frequency of core 0 to f_max;
7          execute / simulate;
8          T_fmax = time taken to execute A_cur;
9          for each frequency f_cur in F, other than f_max do
10             set the frequency of core 0 to f_cur;
11             execute / simulate;
12             T_cur = time taken to execute A_cur;
13             ΔP = ((T_cur − T_fmax) / T_fmax) × 100;
14             save the tuple <A_Score_Acur, GA_Ccur, ΔP, f_cur >;
15         end
16     end
17 end
```

Let us also assume that the processor is capable of operating at N_f different frequencies $F = \{f_0, f_1, \ldots, f_{N_f-1}\}$, with the maximum frequency among these being labeled f_{max}. As discussed earlier, we assume the DVFS can be done on a per-core basis. Algorithm 1 describes how the data collection is done.

3.3 Building the Model

We desire a model that best captures the relationship between a benchmark's memory behavior, that of its co-runners, the frequency at which former is executed, and its performance. Therefore, in the training phase, we use $A\_\texttt{Score}$,GA, $\Delta\texttt{P}$ (as defined in Algorithm 1) and frequency values, as collected in Sect. 3.2 to build the model, as shown in Fig. 3. In the testing phase, the model, given a benchmark, its co-runners, and a desired performance requirement, returns the minimum frequency that guarantees specified performance. There are a variety of machine learning algorithms that can be applied to capture the relationships

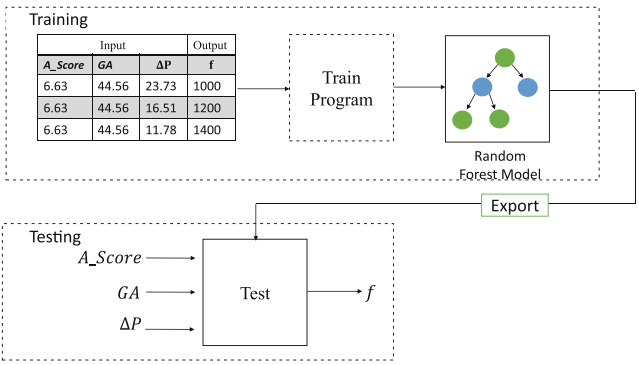


Fig. 3. Training and testing of the model

in different ways. Section 4 discusses the various machine learning algorithms considered, and their respective scores of predictions.

3.4 Application of the Model

The model can be used as a part of *batch* as well as *online* scheduler. In the case of *online* use, at each scheduling decision, for each of the runnable tasks running on the available cores, their previous epochs' performance metrics can be used for tuning each core's frequency. Here, the cost of DVFS transition as well as the time it takes to execute the prediction model should be taken into account to determine the time interval between two consecutive schedules. The input parameters that are collected from performance counters ($A\_\texttt{score}$, GA) will be used along with the QoS policy imposed by the user, like $\Delta P = x$, for $x\%$ of performance loss that the user wants to let go to save energy. Then the model predicts the minimum frequency at which the core should run to satisfy the requested QoS.

4 Comparison of Machine Learning Algorithms

There are a variety of machine learning algorithms to choose from. We used the WEKA (Waikato Environment for Knowledge Analysis) [16] machine learning suite to study the efficacy of the different modeling alternatives. Table 2 shows the list of various machine learning algorithms with their respective correlation indexes when tested using 10 fold cross-validation and our test set data respectively. The experiment was performed on Intel i7-4770 (number of physical cores = 4, logical cores = 8) processor with a speed of 3.4 GHz. We found that the decision tree based `Random Forest (RF)` regression model best captures the relationship between the aggressiveness metrics, the core-frequency, and the performance degradation. In addition to that, the time taken for testing the model is short enough for an online application. `RF` [17] is ensemble of decision trees where each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. The reason that the `RF` model performs well is that it alleviates the overfitting problem, which is common on other regression models.

Table 2. Algorithm comparison

Algorithms	Cross-validation (correlation index)	Test set (correlation index)	Testing time per data point (secs)	Training time (secs)
Linear regression	0.8273	0.8799	2.27 E−05	0.26
MLP regressor	0.9664	0.9815	3.97 E−05	0.17
SVM regressor	0.858	0.9054	2.84 E−05	2.34
REP tree	0.9764	0.9742	3.40 E−05	0.04
Random forest	0.9904	0.9815	3.96 E−05	0.28

5 Evaluation

5.1 Evaluation Setup

We use sniper multicore simulator [19] version 6.0 to validate the proposed model. To model the energy consumption, we use McPAT (Multicore Power, Area, and Timing) integrated power, area, and timing modeling framework [18].

Table 3. System configuration

Core		Caches	
Parameter	Value	Parameter	Value
ISA	X86	L1-D	32 KB, 8-way, WB, 4 cycles
Micro-architecture	Nehalem	L1-I	32 KB, 8-way, 4 cycles
N_C	4	L2(LLC)	2 MB, 16-way, WB, 30 cycles
F	$\{1000, +200, \ldots, 2400\}$ MHz	Cache block size	64
V	$\{0.8, +0.1, \ldots, 1.5\}$ v	Memory latency	45 ns
Technology	45 nm		

The architectural configuration of the simulated system is given in Table 3. We use 45 single-phase, single-threaded, 250 million instruction long SPEC2006 benchmark fragments. The SPEC2006 benchmark suite contains a mixture of compute and memory intensive workloads [21]. The fragments are collected using SimPoint [20].

Data Collection: We use 25 of these 45 benchmarks for the purpose of data collection, that is, set A as defined in Sect. 3.2 ($N_A = 25$). The 25 benchmarks used for the training are selected by their $A\_\texttt{Score}$ values covering the whole range of $A\_\texttt{Score}$ values. We construct the co-runner group C as explained in Sect. 3.2 with $N_{CG} = 10$.

Testing of the Model: We first performed 10-fold cross validation using the training set of 25 applications. We observed a high correlation index of 0.9904, providing a preliminary validation of the proposed model. We also performed online testing of the model. We use the remaining 20 applications for this purpose. We schedule each of the 20 applications on core 0. Let us call this application running on core 0 as the primary application. We randomly select 10 co-runner groups, each group containing 3 applications from this set of 20. These 3 co-runners are scheduled on cores 1–3, and the latter are made to run at the highest frequency setting, that is, 2400 MHz. We evaluate against four different *Quality of Service* (QoS) policies, that is, the maximum degradation in the performance of the primary application that the user is willing to accept. The four

policies are: 5%, 10%, 15%, and 20%. Thus, there were a total of $20 \times 10 \times 4 = 800$ experiments. In each experiment, we use the constructed model to predict the lowest frequency at which core 0 must run such that the QoS policy is honored. Since the output of the model is a real number, we approximate the value to the nearest frequency setting. We then perform a reference run at different frequency settings and select the frequency setting which best satisfies the user QoS policy. We check that if the best frequency matches the predicted one or not. If not, we see by how many frequency steps it deviates. Based on the distribution of the inaccuracies, we calculate the average loss/gain in performance and energy. The results of the testing is presented next.

5.2 Analysis of the Results

Figure 4 shows the average energy saved through DVFS, and the associated average degradation in the performance of the primary application, for the four QoS policies. We see that up to 51% of energy can be saved when the user is willing to sacrifice 20% of the performance.

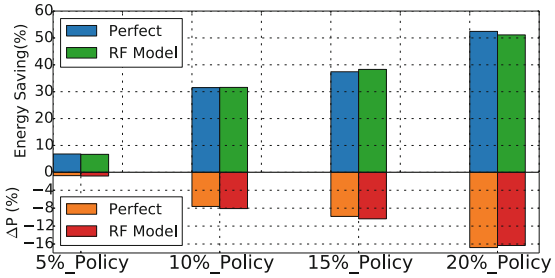
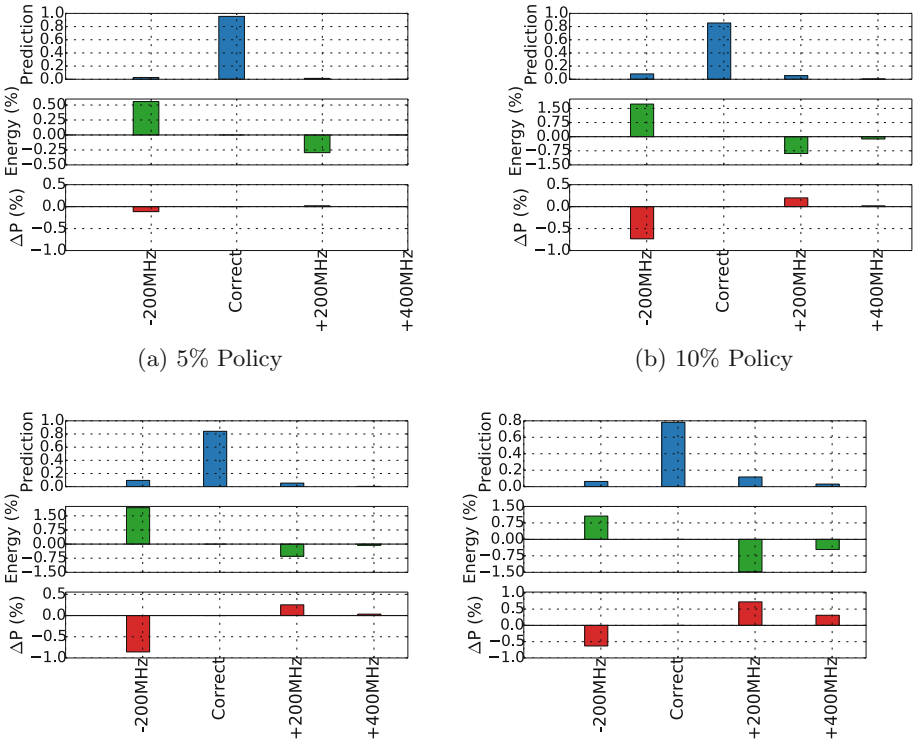


Fig. 4. Performance-energy trade-off for different QoS policies: perfect vs random forest model

We compare our proposed frequency predictor (RF model) against a perfect predictor (one that always predicts the optimal frequency). As can be seen, our predictor performs very close to the perfect predictor, with an accuracy of 1.3%, thereby validating our model – both the choice of the contention metric, as well as the machine learning algorithm. We see that, both with our predictor and the perfect one, in each of the four QoS policies, the observed average performance degradation is much lesser than that specified by the user. This is because of the coarse granularity of the DVFS regulator that allows us to scale the frequency only at steps of 200 MHz.

We further analyze the almost negligible inaccuracies of our model, and describe our findings in Fig. 5. In each of the four figures, the first row depicts the fraction of predictions that were correct (same as the perfect predictor), and the fraction of predictions that were incorrect by -200 MHz, $+200$ MHz, and $+400$ MHz. Note that there were no predictions that were incorrect by a greater



(a) 5% Policy (b) 10% Policy

(c) 15% Policy (d) 20% Policy

Fig. 5. Energy and performance loss/gain because of prediction inaccuracies

margin. The second row depicts the loss/gain in energy savings associated with the incorrect predictions, as compared to the perfect predictor. The third row depicts the associated loss/gain in performance.

We see that the percentage of correct predictions on average is 86%. Among the incorrect predictions, 94% are within one frequency step away from the optimal frequency. When the predicted frequency was higher than optimal, there is a loss in energy savings. Likewise, when the predicted frequency was lower, there is a gain. The net effect of these inaccuracies is negligible, as can be seen in Fig. 4. Additionally, when the predicted frequency is lower than optimal, the QoS policy is not honored. We see that this scenario occurred only in 6.7% of the test runs.

6 Conclusion

In this work, for chips with per-core DVFS capability, with the help of lightweight metrics, we showed that we can predict the application's performance response to shared resource contention and frequency scaling. We also demonstrated that,

given application's and its co-runners' memory behavior just captured through a single parameter of Aggressiveness Score, we can accurately predict the optimal frequency for the given user QoS policy through machine learning. The results demonstrated that on an average, 86% of the predictions were accurate and out of the inaccurate predictions 94% were within a distance of one frequency step (200 MHz). In addition to that, only 6.7% of the total predictions violated the user QoS requirement. In the experiments, we observed that, by allowing 16% of performance degradation, we can save up to 51% core-energy saving. We believe that incorporating additional locality metrics to capture the behavior of the applications can further improve the accuracy of the model. In addition to that, the impact of the model from thermal perspective should also be studied.

References

1. Weiser, M., et al.: Scheduling for reduced CPU energy. USENIX (1994)
2. Zhu, D., Melhem, R., Childers, B.: Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. IEEE TPDS **4**, 686–700 (2003)
3. Cong, J., Gururaj, K.: Energy efficient multiprocessor task scheduling under input-dependent variation. In: DATE 2009, Dresden, Germany (2010)
4. Yao, F., et al.: A scheduling model for reduced CPU energy. In: FOCS 1995 (1995)
5. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: ISLPED 1998 (1998)
6. Kim, S.I., Kim, H.T., Kang, G.S., Kim, J.-K.: Using DVFS and task scheduling algorithms for a hard real-time heterogeneous multicore processor environment. In: EEHPDC 2013 (2013)
7. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. In: ASPLOS 2010 (2010)
8. Abera, S., Balakrishnan, M., Kumar, A.: PLSS: a scheduler for multi-core embedded systems. In: Knoop, J., Karl, W., Schulz, M., Inoue, K., Pionteck, T. (eds.) ARCS 2017. LNCS, vol. 10172, pp. 164–176. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54999-6_13
9. Merkel, A., Stoess, J., Bellosa, F.: Resource-conscious scheduling for energy efficiency on multicore processors. In: EuroSys 2010 (2010)
10. Dhiman, G., Rosing, T.S.: Dynamic voltage frequency scaling for multi-tasking systems using online learning. In: ISLPED 2007 (2007)
11. Khan, U.A., Rinner, B.: Online learning of timeout policies for dynamic power management. ACM-TECS **13**(4), 1–25 (2014)
12. Otoom, M., et al.: Scalable and dynamic global power management for multicore chips. In: ACM 2015 (2015)
13. Ye, R., Xu, Q.: Learning-based power management for multicore processors via idle period manipulation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **33**, 1043–1055 (2014)
14. Islam, F., Lin, M.: A framework for learning based DVFS technique selection and frequency scaling for multi-core real-time systems. In: HPCC 2015 (2015)
15. Shen, H., Qiu, Q.: Contention aware frequency scaling on CMPs with guaranteed quality of service. In: DATE 2014 (2014)
16. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. SIGKDD Explor. **11**, 10–18 (2009)

17. Breiman, L.: Random forest. Mach. Learn. **45**(1), 5–32 (2001)
18. Li, S., et al.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: MICRO 2009 (2009)
19. Sniper Multicore Simulator. http://snipersim.org
20. Calder, B., et al.: SimPoint: picking representative samples to guide simulation (Chap. 7). In: Performance Evaluation and Benchmarking (2005)
21. Jaleel, A.: Memory characterization of workloads using instrumentation-driven simulation. Technical report, VSSAD (2007)