Postprint of article in *Journal of Systems and Software* **85** (4): 960–973 (2012)

EClass: an execution classification approach to improving the energy-efficiency of software via machine learning*

Edward Y.Y. Kan [a], W.K. Chan [b,†], T.H. Tse [a]

[a] Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
[b] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong

A B S T R A C T

Energy efficiency at the software level has gained much attention in the past decade. This paper presents a performance-aware frequency assignment algorithm for reducing processor energy consumption using Dynamic Voltage and Frequency Scaling (DVFS). Existing energy-saving techniques often rely on simplified predictions or domain knowledge to extract energy savings for specialized software (such as multimedia or mobile applications) or hardware (such as NPU or sensor nodes). We present an innovative framework, known as *EClass*, for general-purpose DVFS processors by recognizing short and repetitive utilization patterns efficiently using machine learning. Our algorithm is lightweight and can save up to 52.9% of the energy consumption compared with the classical *PAST* algorithm. It achieves an average savings of 9.1% when compared with an existing online learning algorithm that also utilizes the statistics from the current execution only. We have simulated the algorithms on a cycle-accurate power simulator. Experimental results show that *EClass* can effectively save energy for real life applications that exhibit mixed CPU utilization patterns during executions. Our research challenges an assumption among previous work in the research community that a simple and efficient heuristic should be used to adjust the processor frequency online. Our empirical result shows that the use of an advanced algorithm such as machine learning can not only compensate for the energy needed to run such an algorithm, but also outperforms prior techniques based on the above assumption.

** This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project numbers 111410 and 717811) and a Strategic Research Grant of City University of Hong Kong (project number 7002673).

† Corresponding author.

E-mail addresses: edkan@hotmail.com (E.Y.Y. Kan), wkchan@cs.cityu.edu.hk (W.K. Chan), thtse@cs.hku.hk (T.H. Tse).

Keywords:
DVFS
Workload prediction
Energy optimization
Energy saving
Machine learning

Highlights:

- Adaptively adjusting the processor frequency can conserve energy in program execution.
- Existing techniques are limited to simple heuristics due to the energy concern in computing heuristics.
- We propose *EClass*, a supervised machine learning technique.
- Experimentation shows that *EClass* conserves at least 9.1% more energy than simple heuristic techniques.
- It demonstrates that advanced techniques can outperform simple techniques even though the former consume more energy themselves.

1 Introduction

Unlocking energy efficiency in performing activities is a universal step to deal with global warming and support our digital society. On the other hand, modern software applications have an increasing tendency to interact with various hardware components such as WiFi and GPU, or establish software networks (say, over the Internet) with other application components such as map services at run time. Such a usage trend may increase the resource utilization of a *running* application component. However, the delays incurred when communicating with other components or issuing commands to the underlying operating system followed by receiving the responses may not require a processor to run at a full speed, which provides various opportunities for an energy-aware task scheduling technique [5][11][22] to modify the resources allocated to the application component for completing a program execution.

Dynamic Voltage and Frequency Scaling (*DVFS*) [2] is a technology originally developed for hardware processors that allows a change in processing speed by selecting an alternate processor frequency. For instance, by writing a value to a special-purpose processor register designated for setting up the processor voltage (and hence frequency) to one of the processor-supported configurations, an application may choose to slow down the clock frequency of its servicing processor during idle or low processor utilization periods of

the application. For example, on the Intel XScale platform, a desired processor frequency can be selected by writing a value to the Core Clock Configuration Register (CCCR) and setting the FCS bit of the CCLKCFG register to activate the change in processor frequency [19].

Running a process at a lower frequency may lengthen the overall execution time, which may not result in spending less energy as a whole when compared with setting the processor at a higher frequency to complete the same program execution. Hence, two fundamental problems in setting the above values are to determine (1) when a technique needs to adjust the processor frequency and (2) what the value of processor frequency should be. In general, there are static techniques and dynamic techniques, differing by whether dynamic monitoring of program executions is available.

For ease of presentation, we refer to a set of inputs to a particular program as a *test suite*, and refer to a program that executes a test suite at run time as a *task*. We note that in the real-time systems community, a test suite is often referred to as a workload. Hence, to balance between the two communities, we use the terms test suite and workload interchangeably.

In general, static techniques [4][23][32] analyze the source code of a program, estimate the worst case timing requirements for all possible paths, and use such information to compute frequency values statically for different parts of the program. Such values can be used prior to the execution of the program for any workload. However, static techniques may fail to capture specific characteristics of particular executions of the program. Hence, an underlying processor may not be best tuned for a particular workload. Moreover, it is commonly known that static analysis cannot scale well enough to handle large applications. Dynamic techniques [2][9][35] complement static techniques by monitoring program executions to obtain more precise resource utilization statistics. However, they can only analyze the executions corresponding to a given workload. Based on a monitored portion of an execution, such a technique may apply various policies to predict the best frequency value for the next interval, such as by computing an optimal frequency for the current monitoring interval and deeming it to be the best [14][16]. Dynamic techniques are more scalable than static techniques in handling applications of a practical size. Nevertheless, they may slow down the original execution of a program by incurring execution monitoring and computation of an optimal frequency, and cannot be applied to an API-like framework that cannot be executed by its own. Despite these limitations, dynamic techniques have been widely practiced because of its ability to optimize the energy usage for general program executions, which is particularly attractive when aggressive energy conservation is mandatory. In this paper, we study dynamic techniques.

A test suite for a program can be mapped to a set of program executions. The use of a portion of an execution to predict the frequency value for the remaining portion of the *same* execution may likely result in a locally optimal solution. For instance, for an execution with an increasing demand of computational needs until the very last few instructions of the execution, using a moving average or the latest value of the estimated optimal frequency value based on the monitored portion may be inadequate for predicting the frequencies for the unexecuted portion of the execution. Moreover, to reduce the performance overhead incurred by computing an optimal frequency, many existing techniques [2][16][35] use simple formulas such as doubling or halving the current value to compute a new frequency value based on an existing one. Although such simple formulas may empirically be effective in saving energy, their corresponding techniques do not capture the design rationale of such formulas.

In this paper, we use three major insights to address the above problems in finding a solution.

- First, a typical program is written to serve a particular purpose and is implemented as a set of program modules. Although a program tends to perform distinct activities in different executions, given a specific *subset* of a test suite, the program tends to execute similar sets of *program modules of the same program*, possibly in different orders and with different module occurrence patterns in various execution fragments. For instance, a program must contain a set of basis paths. Although different executions of the program may visit the same program module and run different sequences of statements in this particular module, such executions can be considered as different compositions of these basis paths.

- Second, although different orders and patterns may result in different nonfunctional program behaviors, we observe that even the most I/O-intensive (but typical) programs, such as file management programs, involve some periods of computational-intensive executions. Suppose we define I/O-intensive periods in these programs as less than 25% CPU utilization. In our empirical study, the probability of encountering such an I/O-intensive segment in the most I/O-intensive application is merely 9%, indicating that the probability of encountering k consecutive I/O-intensive segments is only 0.09^k, or 0.00006561 when k is 4, which is a small number. Such a low probability shows that we may use short chains of fragments to model the program processor utilization that covers over 99.9% of all the scenarios encountered.

- Third, the available number (n) of processor frequencies is a small number (such as 11 for XScale in [19]). Because of the presence of computational-intensive and I/O-intensive modules in an interval, in many cases, they lead to the use of highest few and lowest few frequencies, respectively, as the optimal processor frequency. Hence, although there are theoretically n^k possible combinations when modeling all such fragments, since the given value of k is small, the number of combinations is bounded in practice by a small value. Our empirical study confirms this hypothesis. Because of such a small number of combinations, our technique can keep the entire dataset of such values in the memory (rather than storing them in a database or on disk with an excessive amount of memory

page swapping) and locate the optimal frequency by a direct lookup approach. Such a lookup can be efficiently and simply implemented as a hash function provided that one can classify and consolidate utilization values into distinct patterns, and use them as keys to this function. The simplicity further reduces the chance of introducing faults in the implementation, thus enhancing the reliability of the approach.

The net result is a technique that we call *EClass*, standing for **E**xecution **Class**ification. *EClass* is capable of using a small number of utilization patterns extracted from the executions of a small fraction of a workload to characterize almost all the scenarios of program execution fragments. Because of such a characteristic, *EClass* is able to provide a solution that empirically outperforms a solution based on a single execution.

Based on executions on the same platform, *EClass* starts with a training phase for a task. CPU utilization and energy consumption values are collected during training executions in order to determine hardware- and task-specific parameters in the energy model. Utilization values are then clustered and the optimal frequencies for each distinct pattern (of a fixed length) are pre-computed according to the energy model, and stored offline for the frequency assignment algorithm. During online execution, the best frequency is predicted and the current processor frequency is adjusted whenever the execution exhibits a known utilization pattern. The predicted frequency can also be fine-tuned linearly based on the desired execution time. We also adapt *EClass* to develop a complementary technique called *Target*. *Target* treats the execution time of an interval in previous runs of the same test case as if it were the execution time for the same interval in the current run for energy optimization.

We evaluated *EClass* and *Target* on a cycle-accurate power simulator for a realistic hardware platform [17]. We compared them with two existing representative algorithms (which we call *Online* [14] and *PAST* [35]), and a random energy optimization strategy (denoted by *Random*) over three real-life medium-sized subject applications (**minigzip**, **djpeg**, and **jikespg**) from the CSiBE benchmark [15]. Experimental results show that for each subject, the savings achieved by *EClass* and *Target* are always better than that achieved by each of *Online*, *PAST*, and *Random*, and on average, *EClass* and *Target* can save 9.1% more energy in absolute terms compared with the peer techniques, and up to 52.9% savings for specific applications. The difference in energy savings between *EClass* and *Target* is less than 2.2% in all cases because *EClass* is able to operate in unseen executions whereas *Target* cannot. The evaluation results show that in terms of energy savings, *EClass* is the best among all five techniques studied in the experiment. The empirical results are inspiring in that although pattern classification can be much less efficient (and hence requiring more energy) than simply setting a value as in *Online*, this extra cost can be well-compensated at the end.

The main contribution of this work is threefold: (1) It demonstrates the use of a machine learning approach to effectively characterize program executions without excessive training data. (2) It proposes two new online strategies, namely *EClass* and *Target*, which efficiently use such characterizations to predict optimal frequencies for execution fragments. (3) It reports an experiment that verifies the above insights and the proposed techniques. Our experimental results demonstrate that energy-awareness research can benefit significantly from advanced or search-based algorithms.

The rest of the paper is organized as follows: Section II gives a motivating example. Section III describes our models and the details of the algorithm. Section IV presents the experimental results and an evaluation. Section V summarizes related literature on energy optimization. Section VI concludes the paper.

2 Motivating Example

The following is a hypothetical example comparing the computational steps taken by two different frequency assignment algorithms over a dedicated processor capable of running at 200, 266, 333, and 400 MHz. We assume that the algorithm reviews the CPU frequency at the end of each 100 ms interval *j*. The program takes an integer parameter *p* and executes two different branches of the source code, depending on whether *p* is divisible by 3. If it is divisible by 3, the execution lasts for 7 intervals, repeating the instructions corresponding to intervals 1–3 for the duration of intervals 5–7. In another execution where *p* is not divisible by 3, it executes the same instructions corresponding to the first 3 intervals and then terminates. These two executions are detailed in Table 1(a) below:

j	CPU Time @ 733 MHz (ms)	I/O Time (ms)	*PAST* [35] Frequency Prediction (MHz)	Scaled CPU Time (ms)	Scaled Utilization Rate
1	17	83	200	62	43%
2	16	84	200	59	41%
3	39	61	200	142	70%
4	91	9	266	252	97%
5	Repeat interval 1		333	37	31%
6	Repeat interval 2		333	35	30%
7	Repeat interval 3		333	97	63%

(a) Sample executions of a program using *PAST*

j	CPU Time @ 733 MHz (ms)	I/O (ms)	Optimal Freq (MHz)	Utilization Class
1	17	83	266	Low
2	16	84	266	Low
3	39	61	333	Medium
4	91	9	400	High
5	Repeat interval 1		266	Low
6	Repeat interval 2		266	Low
7	Repeat interval 3		333	Medium

(b) Sample executions of a program using *EClass*

Table 1. Illustrative Example

Let us take a look at the scenarios in Table 1(a): Consider a simplified version of the *PAST* algorithm [35], which starts

at the lowest frequency of 200 MHz, increases the frequency by 20% if the current CPU utilization rate (that is, CPU time of past period / (CPU + I/O time)) exceeds 70%, and decreases the frequency by 60% minus the current utilization if the current utilization drops below 50%. For an execution with $p = 3$, *PAST* stays at the lowest frequency for the first three intervals since the utilization after scaling to 200 MHz falls between 50% and 70%. At the end of the third interval, the utilization exceeds 70%. The frequency is increased to 266 MHz in the fourth interval, which is executed only when the parameter p is divisible by 3. The frequency is further increased to 333 MHz for the rest of the execution repeating the instructions corresponding to intervals 1–3.

The frequencies chosen by *PAST* in this example reveal two generic problems inherent in algorithms of its kind. First, the chosen frequency for intervals 1–3 is inconsistent with the frequency for intervals 5–7 even though the same set of instructions and statistics are observed. Second, the choice of frequency can only be the same as the previous interval or one step from it. As the frequency of the previous interval may have no relationship with the optimal frequency of the next interval, the frequency change may go in the wrong direction (as in interval 5 where the frequency should have matched interval 1), or in the right direction but is too subtle (compared with interval 4 of Table 1(b) discussed below).

Table 1(b) shows the same example using an execution with $p = 3$ as an offline training instance for *EClass*. Here, we compute the optimal frequency and classify the utilization for each period using the empirically determined energy model[1] and a clustering algorithm explained in detail in the next section. In this example, we assume that *EClass* has been setup to analyze two intervals of runtime statistics at a time. We refer to this configuration as a sliding window of size $w = 2$.

As shown in the last column of Table 1(b), *EClass* evaluates the sequential changes in utilization classification every two intervals and produces the following rules:

(1) Execute the first w intervals ($w = 2$) at 266 MHz to collect more runtime statistics for future intervals.
(2) After the first w intervals, if the pattern ⟨Low, Low⟩ is matched, *EClass* predicts that the next interval will fall into the "Medium" utilization class. It then sets the frequency to 333 MHz (that is, use 333 MHz after 2 consecutive intervals of "Low" utilization class).
(3) If the pattern ⟨Low, Medium⟩ is matched, *EClass* predicts that the next interval will fall into the "High" utilization class. It then sets the frequency to 400 MHz.
(4) If the pattern ⟨Medium, High⟩ is matched, *EClass* predicts that the next interval will fall into the "Low" utilization class. It then sets the frequency to 266 MHz.
(5) If the pattern ⟨High, Low⟩ is matched, *EClass* predicts that the next interval will fall into the "Low" utilization class. It then sets the frequency to 266 MHz.

[1] For the purpose of illustration, we assume the following parameters for the energy model from Section IV: $P_{off} = 0.35$, $C = 1$, and $m = 3$.

In this scenario, *EClass* addresses the previous two problems by defining rules based on the analysis of utilization patterns for prior executions and the corresponding optimal frequencies. The choice of frequencies is consistent and optimal as per the training instance. We note that this illustrative example cannot represent all possible executions in face of the many hardware and software variables that affect utilization behavior. However, it serves to demonstrate our design rationale supported by the experimental results in subsequent sections.

Moreover, as there are nine possible permutations of Low, Medium, and High to form a sequence of two, *EClass* can produce four such sequences rather than always exhausting all possibilities. This reduced amount of space further allows the implementation to keep the data structure in the memory in the form of a hash function, which provides efficient lookup for the runtime prediction phase of *EClass*.

3 Models and Algorithm

3.1 Preliminaries

In this section, we review the latency model proposed in [9]. A task is a sequence of instructions I whose execution causes some latency and energy dissipation. The number of instructions in I is denoted by $|I|$. Different types of instructions may require different numbers of CPU cycles to execute. We define *CPI* as the mean number of CPU cycles required per instruction for the execution of I. Frequency is measured in hertz (Hz) or cycles per second. To understand the effects of CPU frequency changes on latency and energy spending, we partition the total execution time t of I into two components: on-chip and off-chip, as shown in equation (1).

$$t = t_{\text{on}} + t_{\text{off}} \tag{1}$$

where t_{on} is the on-chip execution time spent within the CPU including data dependencies, fetch dependencies, cache/TLB hits, and branch prediction, and its value is affected by the frequency of the processor; and t_{off} is the off-chip execution time spent on non-CPU activities such as memory access due to cache/TLB misses, and other components synchronized to the bus clock. t_{off} is *not* affected by the CPU frequency. While this breakdown can be inexact for processors that support instruction-level parallelism and out-of-order executions due to the overlapping of t_{on} and t_{off}, the resulting error in equation (1) is small relative to memory access latency, which is about two orders of magnitude greater than the latency typically associated with the execution of a CPU instruction [9].

As explained in Section 1, DVFS is achieved by writing to special processor registers that switch the CPU voltage and frequency to one of the supported configurations. The core frequency is expressed as a nonzero multiple of the external clock frequency (66.667 MHz in our evaluation). For Intel XScale processors, the set Z of available multiples is {3, 4, …, 10, 11} [19]. In other words, eleven distinct core freq-

uencies are supported on this platform from 200 MHz to 733 MHz.

Since t_{off} is unaffected by processor frequency, we may, in the model, simply consider the effects of frequency on t_{on} and the overall latency in finding the optimal frequency. Such latency can be calculated by multiplying the number of instructions by the mean number of CPU cycles required per instruction, and then dividing the value by the frequency used. For instance, let $t_{\text{on}}^{\max}$ be the on-chip latency when the processor runs at full speed $f^{\max}$. Then, $t_{\text{on}}^{\max}$ can be computed as $(|I|.CPI)/f^{\max}$. The on-chip execution time for the instruction sequence I at a *reduced* frequency f^R can be deduced by $t_{\text{on}}(f^R, f^{\max})$ as follows:

$$\begin{aligned} t_{\text{on}}(f^R, f^{\max}) &= \frac{|I|.CPI}{f^R} = \frac{t_{\text{on}}^{\max} f^{\max}}{f^R} \\ &= t_{\text{on}}^{\max} / \left(\frac{f^R}{f^{\max}}\right), \end{aligned} \tag{2}$$

which can be further simplified to

$$t_{\text{on}}(f) = \frac{t_{\text{on}}^{\max}}{f}, \text{ where } f = \frac{f^R}{f^{\max}} \tag{2'}$$

Equation (2) shows that in the model, the on-chip execution time varies linearly with the CPU frequency (which agrees with other authors such as Burd and Brodersen [7]). Substituting $t_{on}(f)$ from equation (2′) into equation (1), we obtain

$$t(f) = \frac{t_{\text{on}}^{\max}}{f} + t_{\text{off}}, \tag{3}$$

which is a parametric equation with parameter *f*.

In order to achieve optimal energy efficiency for different execution hardware, our frequency computation is adaptive to the actual power model of the underlying hardware. Suppose that *P* is the instantaneous electrical power, typically in watts (W). Given the duration of a task execution, the corresponding energy consumption can be computed as $P \cdot t(f)$. To model energy consumption of frequency-dependent components explicitly, we also partition *P* into on-chip and off-chip components, and hence the system-wide energy consumption *E* for a task *T* is defined by equation (4), which is similar to other studies [20][38].

$$\begin{aligned} E(f, t_{\text{on}}, t_{\text{off}}) &= [P_{\text{on}} + P_{\text{off}}]t(f) \\ &= [P_{\text{off}} + Cf^m]\left(\frac{t_{\text{on}}^{\max}}{f} + t_{\text{off}}\right). \end{aligned} \tag{4}$$

where the terms are described in Table 2:

Table 2. Energy Model Description

Term	Description
P_{on}	Frequency-dependent power consumed on-chip
P_{off}	Static power consumed off-chip
f	Processor frequency normalized to 1 at full speed
$t(f)$	Overall latency
C	Effective switching capacitance per processor clock cycle
m	Dynamic power exponent

In CMOS based processors, power dissipation corresponding to P_{on} occurs when charging and recharging internal capacitors at every gate [11]. It is characterized by *C*, the average switching capacitance per clock cycle, and is exponentially related to *f*, where the exponent *m* is between 2 and 3 depending on the hardware architecture. Without loss of generality, we assume $m \geq 2$ without fixing its value in subsequent derivations. P_{off} and *C* are task-dependent and can be determined empirically. We will describe the estimation process of these parameters in Section 4. Note that similar models are used in [20] and [38], but sometimes P_{off} is further decomposed into two terms, depending on whether the components can be put into sleep mode. In our model, we do not distinguish these two kinds of components because they are not relevant to the basic idea of this paper, which is to utilize a classification approach to predict the optimal frequency for the next interval of an execution.

For given task-specific latency parameters $t_{\text{on}}^{\max}$ and t_{off}, we would like to determine the most energy-efficient frequency *f*, which is referred to as the *optimal frequency* in this paper and denoted by *f**. In order to determine *f**, let us differentiate $E(f, t_{\text{on}}, t_{\text{off}})$ in equation (4) with respect to *f*:

$$\begin{aligned} \frac{dE}{df} = \; & mf^{m-1}Ct_{\text{off}} \\ & +(m-1)f^{m-2}Ct_{\text{on}}^{\max} - \frac{t_{\text{on}}^{\max} P_{\text{off}}}{f^2}. \end{aligned} \tag{5}$$

*f** can be computed by setting *dE*/*df* to zero and solving for *f*. Depending on the value of *m*, analytical solutions may be found. In our *EClass* technique, we solve equation (5) using the Newton-Cauchy framework [13], which runs efficiently with the discrete and limited domain of *f*.

Taking the second derivative of $E(f, t_{\text{on}}, t_{\text{off}})$, we obtain

$$\begin{aligned} \frac{d^2E}{df^2} = \; & m(m-1)f^{m-2}Ct_{\text{off}} \\ & +(m-1)(m-2)f^{m-3}Ct_{\text{on}}^{\max} + \frac{2t_{\text{on}}^{\max} P_{\text{off}}}{f^3}. \end{aligned} \tag{6}$$

For $m \geq 2$, $f > 0$, $C > 0$, $t_{\text{on}}^{\max} > 0$, and $t_{\text{off}} \geq 0$, the second derivative is always nonnegative, which verifies that $E(f, t_{\text{on}}, t_{\text{off}})$ is a convex curve and has a minimum value when $dE/df = 0$

3.2 EClass: Our Framework

In this section, we present our technique ***EClass***, which is a classification-based approach consisting of two phases: *offline training* and *runtime prediction*.

3.2.1 Offline Training Phase

In this phase, *EClass* collects the execution statistics of sample runs of a task. Suppose that there is a set of test cases *TC*. *EClass* executes every test case *t* in *TC* by setting its underlying processor at f^{max}. It results in a set of executions *R*. *EClass* divides each execution $R_i \in R$ into a consecutive sequence of intervals, and collects a vector *V* of performance attribute values (such as the amount of memory used and the time taken for the interval) at the end of each interval. Let l_i denote the number of intervals for R_i, $s_{i,j}$ denote the values of

V collected at the end of the j-th interval during the execution of R_i, and S denote the set of all such $s_{i,j}$ collected.

In our experiment to be presented in Section 4, we collect the set of attributes that are useful for the estimation of t_{on} and t_{off}, and are available to the simulator. Note that, in principle, there is no restriction on the choice of statistics to be collected.

EClass then divides S into subsets using a partition clustering algorithm, and the optimal frequency for each interval, denoted as $f^*_{i,j}$, is computed from each value of $s_{i,j}$ based on equation (5). The purpose of discretization is to identify major changes in utilization. Specifically, we define $\Gamma: S \rightarrow 2^S \setminus \emptyset$ as a classification function that maps each $s_{i,j}$ to one of the nonempty subsets of S. (In our experiment, we apply the Expectation-Maximization (EM) algorithm to implement Γ.) We denote $\Gamma(S_{i,j})$ by $C_{i,j}$. After applying Γ, *EClass* transforms each training run $R_i = \langle r_{i,1}, r_{i,2}, \ldots, r_{i,l_i} \rangle$ into a series of couples $\langle (C_{i,1}, f^*_{i,1}), (C_{i,2}, f^*_{i,2}), \ldots, (C_{i,l_i}, f^*_{i,l_i}) \rangle$. We denote the set of such series by R^*.

Table 3 extends the example in Table 1(b) with utilization values for three executions with different parameters p after the *training* phase.

3.2.2 Runtime Prediction

The *prediction* phase of *EClass* is performed online. Similar to the *training* phase, the performance statistics of V are collected at regular intervals. Specifically, for every w intervals (where w denotes the size of the sliding window), the optimal frequency for the next interval is predicted by comparing the statistics collected within the current window period with that in R^*.

Table 3. Sample Records for Training Runs

p	Records in R^*					*Actual frequency corresponding to $f^*_{i,j}$ (in MHz)*	Intuitive meaning (in terms of CPU utilization)
	i (execution)	j (interval)	$S_{i,j}$ (e.g., CPU time)	$C_{i,j}$	$f^*_{i,j}$		
3	1	1	17	C1	0.36	266	Low
		2	16	C1	0.36	266	Low
		3	39	C2	0.45	333	Medium
		4	91	C3	0.55	400	High
		5	17	C1	0.36	266	Low
		6	16	C1	0.36	266	Low
		7	32	C2	0.45	333	Medium
4	2	1	18	C1	0.36	266	Low
		2	17	C1	0.36	266	Low
		3	35	C2	0.45	333	Medium
5	3	1	19	C1	0.36	266	Low
		2	16	C1	0.36	266	Low
		3	27	C2	0.45	266	Medium

The basic idea is to search for a consecutive subsequence $\langle C_{i,j}, C_{i,j+1}, \ldots, C_{i,j+w} \rangle$ of R_i matching the actual utilization detected in this phase, and then use $f^*_{i,j+w+1}$ as the predicted value. If multiple matching subsequences exist, the most probable frequency is used as the prediction. If two different frequencies have the same probability, one of the frequencies is chosen arbitrarily.

Formally, we let $C' = \langle \Gamma(s'_1), \Gamma(s'_2), \ldots, \Gamma(s'_w) \rangle = \langle C'_1, C'_2, \ldots, C'_w \rangle$ be the sequence of statistics of V collected and then classified by Γ during an actual execution. The function Φ is defined as

$$\Phi(R_{i,j}, C', f) = \begin{cases} 1 & \text{if } C_{i,j} = C'_1 \wedge \cdots \wedge C_{i,j+w} = C'_w \wedge f = f^*_{i,j+w+1} \\ 0 & \text{otherwise} \end{cases}$$

for $i = 1, 2, \ldots, |R|,\ j = 1, 2, \ldots, l_i - w - 1$.

The frequency prediction for C' is

$$\text{argmax}_f \sum_i \sum_j \Phi(R_{i,j}, C', f) \quad (7)$$

If more than one value satisfies equation (7), f is chosen arbitrarily from one of such values.

In the example presented in Table 3, assuming that $w = 2$, the sequences of clustering results for R_1, R_2, and R_3 are $\langle C1, C1, C2, C3, C1, C1, C2 \rangle$, $\langle C1, C1, C2 \rangle$, and $\langle C1, C1, C2 \rangle$, respectively. Suppose the actual collected statistics $C' = \langle \Gamma(18), \Gamma(17) \rangle = \langle C1, C1 \rangle$. The subsequence exists consecutively in all training runs with two possible frequency predictions $f^*_{1,3} = f^*_{1,7} = f^*_{2,3} = 333$ and $f^*_{3,3} = 266$. Because the frequency of 333 appears twice in R_1 and R_2, it maximizes the Φ function in equation (7), and is therefore the final prediction. If no consecutive subsequence in R_i is found to match C', the frequency is unchanged to avoid the overhead of switching frequencies without sufficient justification. (We note that, in the absence of a matching subsequence, one may employ a hybrid technique such as running *PAST*. Such a strategy can give additional insights on top of *EClass*). At the beginning of the task execution when the interval count is less than w, the prediction model simply suggests the most frequently occurring frequency for that interval (i.e., the statistics mode). In the running example, the model suggests the mode of $f^*_{1,1}$, $f^*_{2,1}$, and $f^*_{3,1}$, which is 266 for the first interval.

Further optimization. After classifying all the training executions, further optimization can be achieved by exploiting the predicted optimal frequencies of a subsequence of future intervals ($f^*_{i,k}$, $k = j+w+2, j+w+3, \ldots, l_i$). For instance, if the solution of equation (7) is $f^*_{i,j+w+2}$ for most values of i, we can use this frequency for an additional interval, thus reducing the overhead of solving equation (7) repeatedly.

Flexibility optimization. To increase the flexibility of the implementation of *EClass*, we include a control parameter $\alpha \in [0, 1]$ in the *prediction* phase. This parameter is an intuitive, linear control knob on the execution time, as it varies the execution time from the smallest value running at the highest frequency ($\alpha = 1$) to the largest value running at the most energy efficient frequency ($\alpha = 0$). A similar parameter has been used in [9] to control the amount of energy savings corresponding to a performance loss of roughly ($1-\alpha$) in terms of execution time.

From equations (1) and (2), the adjusted execution time $t(\alpha)$ is

$$t(\alpha) = \frac{t_{\text{on}}^{\max}}{f} + t_{\text{off}} = \alpha\left(t_{\text{on}}^{\max} - \frac{t_{\text{on}}^{\max}}{f^*}\right) + \frac{t_{\text{on}}^{\max}}{f^*} + t_{\text{off}}$$

Solving for the frequency f that achieves $t(\alpha)$, the frequency prediction can be expressed as

$$f = \frac{f^*}{\alpha(f^* - 1) + 1} \tag{8}$$

3.3 EClass Algorithm

To summarize the steps in both phases, the proposed frequency assignment algorithm *EClass* is shown as follows:

Algorithm 1. Frequency Assignment Algorithm *EClass*

```
1:  Γ, R* ← the cluster algorithm and the training data from the training
        phase
2:  M ← ⟨mode(f*_1), mode(f*_2), …, mode(f*_{w−1})⟩
3:  C' ← ∅
4:  call updateFreq(1) at the first interval

5:  procedure updateFreq(k)
6:      s' ← collect statistics for the past k intervals
7:      if k > w then C' ← C' \ C'(1) end if
8:      C' ← C' ∪ ⟨Γ(s')⟩
9:      if k < w then f ← M(k)
10:     else
11:         f ← solve equations (7) and (8) for the prediction
12:         if f = 0 then f ← getProcessorFrequency() end if
13:     end if
14:     if f ≠ getProcessorFrequency() then setProcessorFrequency(f)
        end if
15:     call updateFreq(k+1) at the next interval
    end procedure
```

Lines 1–3 initialize the algorithm for a task. In line 2, mode(f^*_i) represents the most frequently occurring value in $\{f^*_{1,i}, f^*_{2,i}, \ldots\}$. The procedure in line 5 is called at regular intervals. Runtime statistics are collected in lines 6–8 for the sliding window of w intervals. Due to possible frequency changes, the intervals are defined by cycles rather than by time. In line 9, if the length of C' is insufficient for prediction, the mode value corresponding to the interval count is returned as the target frequency. The oldest statistics are shifted out at every call since only w sets of statistics are required by the model to make a frequency prediction. Lines 9–12 utilize the prediction model as described in the previous subsection. As explained earlier, a hash function is used to efficiently lookup the predicted frequency in this step. We first convert C' into a string with each classification represented by an ASCII character. For instance, the characters A–C represent classifications C1–C3 in Table 3. The string "ABBC" thus represents the sequence $\langle C1, C2, C2, C3\rangle$. We then use this string representation as the hash key to look up the desired frequency already mapped during the *training* phase. Line 14 updates the processor frequency if necessary. Note that the retrieval and setting of the current processor frequency to f are denoted by *getProcessorFrequency* and *setProcessorFrequency*(f), respectively.

4 Evaluation

4.1 Research Questions

In this section, we aim to answer three research questions to validate *EClass*.

RQ1: Does *EClass* save more energy than existing techniques?

RQ2: To what extent may *EClass* compete with existing techniques without extensive training?

RQ3: To what extent can *EClass* be efficient and flexible in the tradeoff between performance and energy savings?

4.2 Experimental Setup

To accurately measure energy consumption, we adopted XEEMU [17], a cycle-accurate power simulator that simulates executions of C programs on XScale processors. According to [17], the simulator achieves accurate energy consumption estimation to within 1.6% error on average compared with real hardware. One of the major advantages of using an accurate simulator rather than physical hardware is that it improves the reproducibility of the experimental results and facilitates fair comparisons with future research efforts. The frequency assignment algorithms are implemented in Java, and executed once for each test case over a dedicated instance of the simulator so that the statistics of each simulation correspond entirely and specifically to the execution of the test case.

We extracted three types of performance attribute values V from the XEEMU simulator as parameters to our energy and prediction models: **sim_cycle_***, **sim_memory_dep**, and **sim_diss_***. First of all, **sim_cycle_*** is the total simulation time in cycles per frequency. As explained before, the simulation time depends on frequency; this counter is an obvious choice to indicate the end of each interval for all algorithms in terms of the number of elapsed cycles. **sim_memory_dep** is the total number of stall cycles caused solely by the memory, which matches the definition of t_{off} in equation (1). Similar events related to data stalls were used in other studies [11] and were shown to be effective for the estimation of on-chip and off-chip duration and power consumption. **sim_diss_*** constitutes the basis of comparison between the algorithms. It is the total power dissipation per frequency (in watts), including power consumed by the processor and the memory. The reading represents the total power ($P_{\text{off}} + P_{\text{on}}$) in equation (4). We have reconfirmed our understanding of this counter by observing that the total energy consumption reported by the simulator always matches the sum of the values of **sim_diss_*** multiplied by **sim_cycle_***.

We modified the source code of the simulator so that it would pause at the end of each interval, output the running values of the first two statistics, and await new frequency settings (if any). In the *training* phase, $f^*_{i,j}$ is computed by solving equation (5) with m = 3 using the roots function in MATLAB. Note that the authors of [7][29][38][39] also

assumed cubic power models. If the computed solution lies between two frequencies supported by XEEMU, the more energy-efficient of the two (according to equation (4)) is used. We note that further energy savings may be achieved by running each frequency for part of the duration of the interval [29], but we have adopted the simpler approach to avoid additional overhead. The classification function Γ is initialized in the Expectation Maximization (EM) algorithm with automatic selection of the number of clusters n using cross-validation. The primary advantage of our choice of implementation over other clustering algorithms is that it is well-established with a strong statistical basis, runs in linear time with respect to input size, and converges fast [12]. Clustering and classification is done using the EM clustering algorithm provided in Weka [36].

4.3 Task-Specific Parameters

As discussed in the previous subsection, the statistics **sim_diss_*** represents $P_{\text{off}} + P_{\text{on}}$ in equation (4), but in order to solve equation (5) in the *training* phase, the task-specific parameters P_{off} and C must be determined. Because there are no performance monitors in our test bed that can map individually to these parameters, we need to estimate their values for all frequencies using the energy consumption statistics and execution times collected from training runs. For each task, suppose n test cases of the task are executed at q different supported speeds $f_1, f_2, \ldots, f_q$. We obtain the following $n \times q$ (≥ 2) equations by expanding equation (4):

$$E_{i,f} = t_{i,f} P_{\text{off}} + t_{i,f} C f^m, \quad i = 1, 2, \ldots, n \text{ and } f \in \{f_1, f_2, \ldots, f_q\}$$

where $E_{i,f}$ is the energy consumption of the i-th instance at frequency f, and $t_{i,f}$ is the total time spent for that execution. These equations can be rewritten in matrix form:

$$Ax = b \tag{9}$$

where $A \in \mathbb{R}^{nq \times 2}$, $x \in \mathbb{R}^2$, and $b \in \mathbb{R}^{nq}$, thus:

$$\begin{pmatrix} t_{1,1} & t_{1,1} f_1^m \\ \vdots & \vdots \\ t_{1,q} & t_{1,q} f_q^m \\ \vdots & \vdots \\ t_{n,1} & t_{n,1} f_1^m \\ \vdots & \vdots \\ t_{n,q} & t_{n,q} f_q^m \end{pmatrix} \begin{pmatrix} P_{\text{off}} \\ C \end{pmatrix} = \begin{pmatrix} E_{1,1} \\ \vdots \\ E_{1,q} \\ \vdots \\ E_{n,1} \\ \vdots \\ E_{n,q} \end{pmatrix} \tag{9'}$$

This system of equations is over-determined because there are nq equations with only two unknowns. Although it is very likely that there is no distinct vector x that will satisfy all of the equations, it is possible to find x that minimizes error ($\|AX - B\|$) by rewriting equation (9) as follows (after [3]):

$$x = (A^T A)^{-1} A^T b \tag{10}$$

It is not efficient to solve for $Ax = b$ using equation (10), but we note that its solution is equivalent to fitting linear regression models in the form $Ax = b$ for different frequencies in equation (9′) while minimizing the least square errors. As a result, we used the matrix left division function in MATLAB to solve for $x = (P_{\text{off}}, C)$ in our implementation, which computes the least squares solution using an efficient QR decomposition algorithm.

4.4 Peer Techniques for Comparison

To evaluate the effectiveness of *EClass* in conserving energy, we have faithfully implemented in Java the algorithms *PAST* [35] and *Online* [14]. For ease of reproducing our results, we also report on the specific parameters used by these two algorithms. In the implementation of *PAST*, we defined CPU utilization as u, and *excess* as *on-chip time* $\times (1 - f)$. Following the frequency adjustment parameters in [35], f is set to 1 if *excess* > *off-chip time*, increased by 20% if $u > 70\%$, and decreased by $60\% - u$ if $u < 50\%$. f is mapped to the supported frequencies as described above. In the implementation of *Online*, the penalty for data cache miss (PEN) is set to two cycles since any higher value causes negative μ values in the algorithm presented in [14]. The frequency adjustment interval for both algorithms is set to match the original literature (10 ms) to facilitate realistic comparison.

In our proposed algorithm *EClass*, the interval is cycle-based. As our simulator is for XScale processors, we arbitrarily set the interval length to one million cycles and $w = 4$ for the size of the sliding window in the prediction model. We have conducted initial trials to observe the effect of setting different values of the interval length, and observed that different tasks and execution cases may favor shorter or longer intervals, but there is no definite advantage of one over another. We note that the effects of interval lengths and window sizes may be affected by the actual implementation. For instance, the interval length may be set as a multiple of the scheduler quantum if the technique is implemented in the OS kernel. Such effects may deserve further investigation in a separate study. In any case, the choice of one million cycles can draw meaningful comparisons in our experiment because f^* is found consistently to be one of the three lowest available frequencies for the test cases of all subjects with different interval lengths. The overheads resulting from window sizes will be discussed in Section 4.8.

To evaluate the effectiveness of the computed f^* in *EClass*, we also included the results for running each training at f^* for all intervals (denoted by *Target*), and a hypothetical *Random* algorithm, which is identical to *EClass* except that it assigns one of the supported frequencies randomly instead of using the computed f^* in line 11 of Algorithm 1.

Because merely selecting a frequency randomly may already have energy saving effect, *Random* does not only select an available frequency randomly. As described above, it reuses the implementation of *EClass* and merely nullifies the effect of the computed frequency by randomly reporting one of the available frequencies. This technique can serve as the lower bound of whether a classification approach (rather than

the implementation) may have any effect in achieving energy savings. In particular, the difference in energy savings between *EClass* and *Random* can be used to quantify the net effect gained by our effective classification approach.

Target can be useful in rerunning test cases on the same program. In such circumstances, the statistics profile can be reused for the reruns of the test cases.

4.5 Subject Applications

Three subject applications are selected from the CSiBE benchmark environment [15]: **minigzip** (version zlib 1.1.4), **djpeg** (6b), and **jikespg** (1.3). The scale of our experiment is comparable to those in some of the previous publications [2][14][21], which are evaluated on 2 to 4 related subjects under various configurations. While both synthetic benchmarks [2] and real-life inputs (such as 8 movie fragments of roughly 20 s each in [21]) are used in these evaluations, we have opted to use real-life inputs to ensure the applicability of our results, and repeated each experiment with a considerable input size (e.g., 1000+ files for **minigzip**) that can be completed in reasonable time. The subject applications have been tested to run properly in the XEEMU simulator [17], and have varying computation and I/O intensity so that the algorithms can be put under a fair comparison with different patterns of CPU utilization. The applications are written in standard C and are compiled for XEEMU using the Wasabi cross compiler tool chain [34] with optimization option –O3. **minigzip** (with 5576 lines of code[2]) is mostly memory-dominant, but its execution is nonetheless affected by the compressibility of the input. In our experiment, we found that moderately compressible input resulted in maximum latency with respect to the input size. On the other hand, **djpeg** (with 19500 lines of code), is a computation-intensive JPEG decompressor that utilizes many shift operations. **jikespg** (with 18064 lines of code) is a parser generator for the Jikes compiler that contains both computation and memory-dominant regions [17].

4.6 Procedures and Test Cases

In the experiment, **minigzip** was used to compress plain text, images, and binary files in order to provide varying degrees of compressibility. **djpeg** was used to decompress JPEG images into PBMPLUS format in the experiment (which is also its typical use). **jikespg** was used to process the language grammar files for the Java language versions 1.4 and 1.5. We have verified that these input files work properly with the application, and are publicly available on the Internet.

To enhance the applicability of the experimental results so that they are relevant to real-life setting, documents and images were downloaded from the ten most popular websites on the Internet [26]. There are a total of 1016 test files (over 100 Mb) for **minigzip**, 806 test files (3.88 Mb) for **djpeg**, and 5 Java language grammar files for **jikespg**. We chose this set of test cases with the belief that they can simulate different utilization patterns while the entire experiment can be completed in reasonable time.

To answer RQ1, we ran *EClass* with the full set of test cases as training data, and then used the same set of test cases for the *prediction* phase and other peer techniques. We applied the same setting to *TARGET* and *Random*. Neither *PAST* nor *Online* requires any training phase. Hence, we simply applied the same test suite to them.

For RQ2, to analyze the reliability of our approach to the *training* phase, *EClass* was executed first with the full set of test cases as training data, and then the same set of test cases was used for the *prediction* phase. We set the control parameter $\alpha = 1.0$ to compare the techniques in terms of energy savings. We also applied the same set of test cases to every other technique.

We then repeated the experiment by randomly selecting 10% of the test cases (with a minimum of 2) for the *training* phase, and applied the remaining 90% of the test cases in the *prediction* phase. To facilitate comparison, α is unchanged and stays at 1.0. We also applied the same set of test cases (that is, the remaining 90%) to every other technique.

To answer RQ3, we measured the hardware access time, predicted fetch time, and energy consumption, as well as the actual number of predictions generated for the sequential matching of utilization classes, which represented the storage and runtime overhead of the *prediction* phase of *EClass*. For the flexibility of performance tuning, we evaluate α in the 10% training / 90% testing experiment. We set α to 0, 0.5, and 1 in separate parts of the experiment, and evaluate its effects against the cumulative execution time as each experiment progresses.

4.7 Effectiveness Metrics

In the subsequent discussion, we evaluate the algorithms in terms of the mean energy consumption of all test cases to determine their effectiveness. The energy needed to compute the optimized frequency values for the algorithms (except for the *training* phase) are also discussed. The results are presented in normalized values such that 100% represents the consumption of the *EClass* algorithm.

4.8 Experimental Results and Discussions

In this section, we analyze the data and answer each research question. We first report the energy savings achieved by *EClass*: In our experiment, when compared with full speed executions without DVFS optimization, *EClass* still achieves 68.1%, 55.8%, and 55.2% energy savings for **minigzip**, **djpeg**, and **jikespg** respectively. As expected, more energy is saved running **minigzip** than **djpeg** because of the memory dominant property of **minigzip**.

RQ1. ***Does EClass save more energy than existing techniques?*** Figure 1 shows the mean energy consumption of

[2] Counted using CLOC [10] for the C programming language excluding blank and comment lines

each application running the algorithms using all the test cases for both training and testing.

Compared with *Online*, *EClass* is able to achieve 9.1% more energy savings on average with a maximum of 12.2% for **djpeg**. In all cases, *EClass* outperforms *Online*. We find that the results on *Target* are 99.2%, 100.0%, and 102.2% of that of *EClass*, respectively. The results of both *EClass* and *Target* show that they can be encouraging alternatives to *Online*.

As for **jikespg**, *PAST* is *significantly* less energy efficient than all other algorithms. In fact, we find that *PAST* executed **jikespg** at full speed most of the time. It shows that the heuristics used in *PAST* has an obvious limitation to a subclass of applications represented by **jikespg.**

Moreover, we measure the difference in energy consumption between *EClass* and *Random*, as indicated by the bar in the middle of each group. The normalized savings are 60.4%, 40.5%, and 46.2%, respectively. The results show that the classification does have predictive ability to conserve energy.

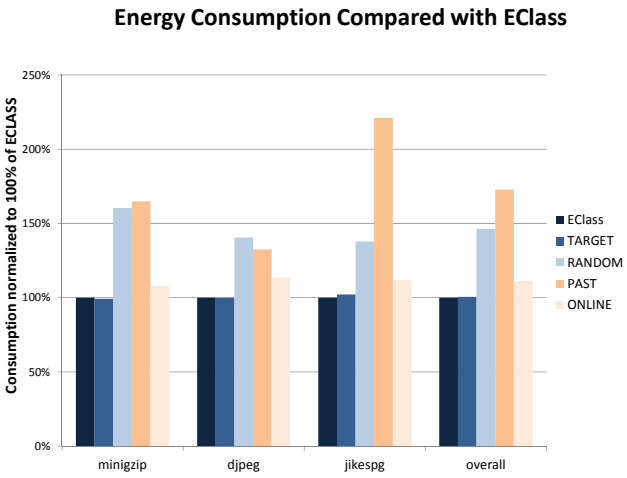


Figure 1. Energy consumptions normalized to 100% of *EClass*

To understand the situation better, we examined the source code of the subjects and the data obtained from the experiment. For a memory dominant application such as **minigzip**, the optimal strategy to conserve energy may be to simply stay at the lowest frequencies. For the case of *Online*, once the current frequency is set to such a value, simply reusing the current frequency for the next interval may already suffice to produce good savings. On the other hand, if the average utilization is higher or more vibrant, the result on **djpeg** shows that it becomes more important to explore utilization patterns and adjust the frequency adaptively.

When compared with the algorithm *Target*, *EClass* performs extremely close to the behavior as if *f** was executed in all the intervals for all test cases. It shows that even if more precise information can be available, the resulting energy savings may not be significantly more than that already achievable by *EClass*. *Target*, on the other hand, is simpler in implementation for its training phase, and its result is more predictable than *EClass* despite the requirement to run every test case once in the *training* phase.

An interesting observation is that *Random* consumes less energy that *PAST* for both **minigzip** and **jikespg**. Such an observation reinforces our intuition that statistics based on the most recent history may not be an ideal energy conservation predictor of future utilization needs.

Table 4 summarizes the mean and standard deviation values in terms of energy consumed. We find that *EClass* and *Target* are the best two techniques in terms of either measure. In essence, we attempt to predict the optimal frequency for an execution. Such a prediction cannot be precise. Let us use the standard deviation values as a rough measure of the accuracy achieved by a technique. Comparing the standard deviations achieved between *Random* and *EClass* shows that the accuracy achieved by *EClass* improves over that of *Random* by 39.0%, 20.1%, and 24.2%, respectively. In fact, as shown in the table, *EClass* can be almost as accurate as *Target* on average. The performance of *PAST* appears to be the least accurate because its achieved values are larger than the ones achieved by *Random*. It reconciles the intuition that we have discussed in the paragraph above.

Table 4. Energy Consumption Summary for 100% Training

	Target	***Random***	***PAST***	***Online***	***EClass***
minigzip					
Mean (J)	0.0112	0.0180	0.0186	0.0121	0.0112
Std. Deviation	0.0025	0.0041	0.0057	0.0026	0.0025
djpeg					
Mean (J)	0.0022	0.0029	0.0036	0.0023	0.0022
Std. Deviation	0.0161	0.0214	0.0313	0.0174	0.0171
jikespg					
Mean (J)	0.2483	0.3272	0.5210	0.2664	0.2408
Std. Deviation	0.1610	0.2076	0.3240	0.1695	0.1574

We note that there are instances where *EClass* slightly outperforms *Target* in terms of energy savings. We have investigated the causes. We conjecture that this is potentially caused by the estimation step in equation (9), resulting in suboptimal predictions for certain test cases. Another potential cause is that the original execution is slightly perturbed by the insertion of frequency switching instructions missing from the training runs.

In conclusion, *EClass* saved the most energy for all three applications compared with the other algorithms studied, and *Target* is slightly more accurate.

RQ2. ***To what extent may EClass compete with existing techniques without extensive training?*** Figure 2 shows the energy consumption in box and whisker plots for each of the algorithms. To make the plots more compact, we use *T* for *Target*, *R* for *Random*, *PT* for *PAST*, *O* for *Online*, and *E* for *EClass*. The results are presented with the median values marked by the (red) line in the middle of each box, and the whiskers extending to 1.5 times the inter-quartile range from the ends of each box. Like the observation on Figure 1, *EClass* and *Target* exhibit highly similar mean and median values.

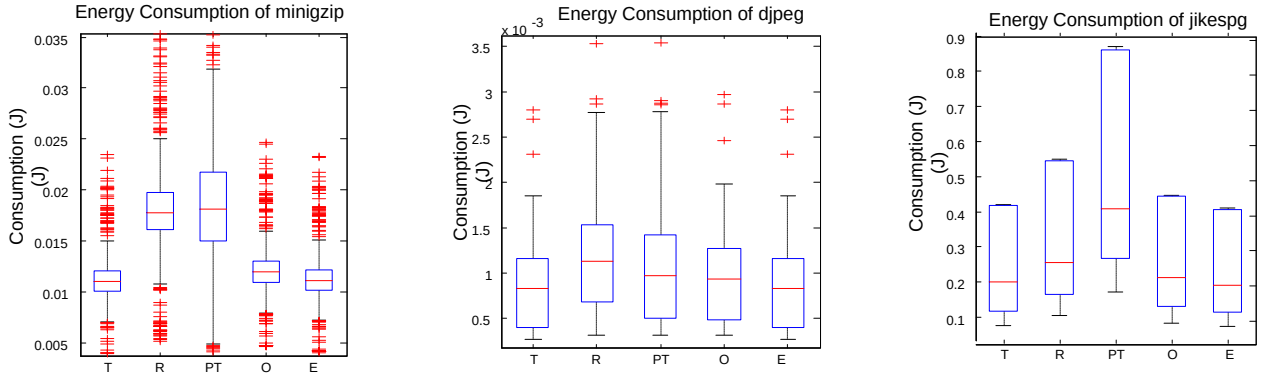
Figure 3 shows the box and whisker plots for the case where only 10% of the test cases (with a minimum of 2) are used for training and the rest for the testing phase.

We note that although the chosen test cases should only affect *Target* and *EClass*, we show the results in Figure 3 using the same subset of test cases for all algorithms (excluding the 10% training cases) to ensure a fair comparison. This explains why Figures 2 and 3 are not identical even for *Random*, *PAST*, and *Online*.

We observe that the results in Figure 3 appear very similar to those in Figure 2 in terms of the shape of the bars. For each test execution, we also plot the normalized energy consumption when using 10% of the test cases for training against that when using 100%. The results are shown in Figure 4. We find that the results have a linear relationship with both the slope and coefficient of determination (R^2) close to 1 for each subject. This means that, with only 10% training, *EClass* is able to achieve energy savings similar to that of 100% training.

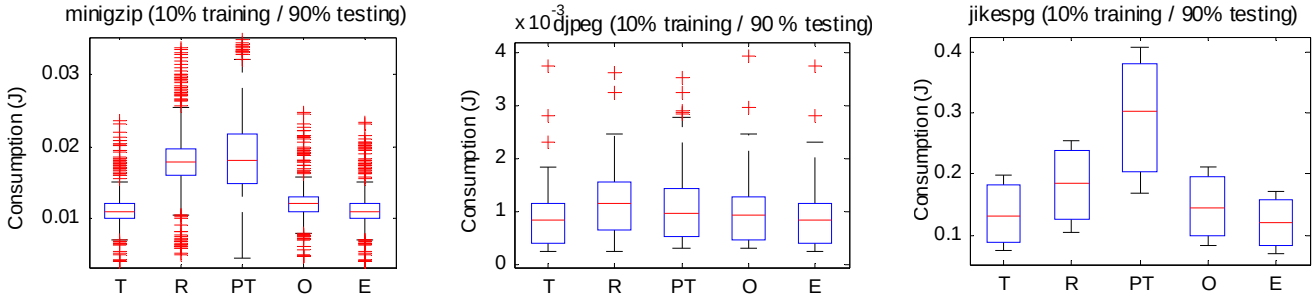
Another insight gained from this observation is that even though *EClass* can be easily augmented by a feedback mechanism to learn new utilization patterns online, such mechanism may not bring much improvement as far as the experiment is concerned because the results of using 10% of test cases for training is already very comparable to that of using 100% of test cases for training.

In general, *EClass* and *Target* result in smaller interquartile ranges for energy consumption, and therefore generate energy savings more consistently. *EClass* also achieve the lowest median values for all three applications. Both *EClass*



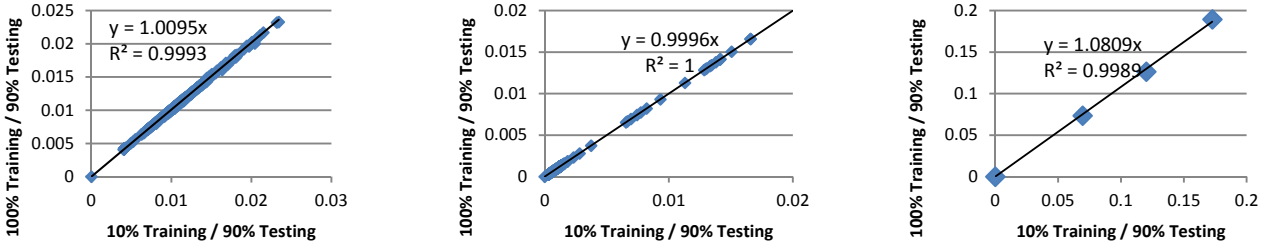
(a) **minigzip** (b) **djpeg** (c) **jikespg**

Figure 2. Energy consumptions on subject programs (with 100% training and 100% testing)



(a) **minigzip** (b) **djpeg** (c) **jikespg**

Figure 3. Energy consumptions on subject programs (with 10% training and 90% testing)



(a) **minigzip** (b) **djpeg** (c) **jikespg**

Figure 4. Effect of *EClass* training on subject programs

and *Target* compete well with existing techniques without extensive training.

RQ3. ***To what extent can EClass be efficient and flexible in the tradeoff between performance and energy savings?*** To answer this question, we evaluate the overheads related to accessing the Performance Monitoring Unit (PMU) to gather online statistics, and the time required to fetch the frequency prediction from the hash function implementation.

The time to access the PMU via read/write is less than 1 µs [9], which can be safely ignored as our prediction interval is in the order of milliseconds. In our experimental setup, timing statistics such as t_{off} can be obtained directly from the PMU. We note that if such statistics on another hardware platform require non-trivial computations from multiple PMU values, additional overheads will be introduced in the training (offline) and prediction (online) phases. In this regard, the complexity of the proposed technique may depend on the availability of PMU counters, while its effectiveness depends on the accuracy of the collected or computed timing statistics. Using XEEMU in our experiment with a hash table containing 7300 predictions and w = 4 (see **minigzip** in Table 5), the measured runtime overhead ranges from 89 to 196 µs depending on the CPU frequency, which translates into 3.9% to 6.5% per interval of one million cycles. In terms of energy, the average overhead per prediction fetch ranges from 0.0207 to 0.0589 mJ, depending on the frequency. This translates into a range from 0.01% overhead for **djpeg** with an average of 4.2 fetches per execution, to 0.15% for **minigzip** with an average of 25.4 fetches per execution.

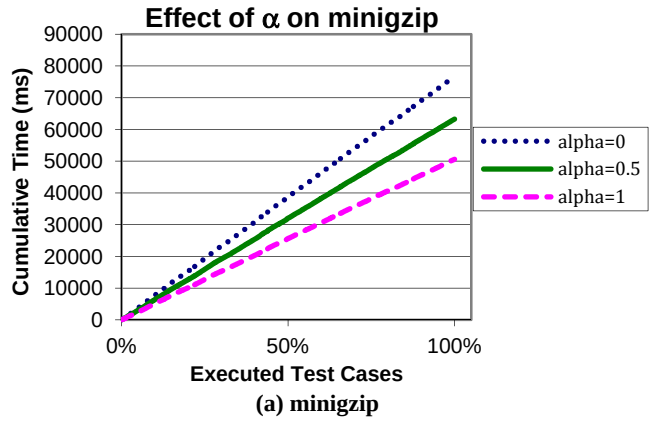
The α value for performance fine-tuning is evaluated in the 10% training / 90% testing part of the experiment. Figure 5 shows the effects of α on the three subject applications. When α is set to 1, the subject applications are always executed at maximum frequency resulting in the least cumulative execution times. When α is set to 0, on the other hand, the most energy-efficient frequency is used and the execution time accumulates the fastest when compared with other values of α. When α is set to 0.5, the cumulative execution time always stays roughly midway in between the lines where $\alpha = 0$ and $\alpha = 1$ for all three applications. This means that the control parameter α introduced in *EClass* controls the tradeoff linearly between performance and energy savings with respect to execution time.

The prediction model in *EClass* can become ineffective if the size w of the sliding window is too small. It may be necessary to increase the window size for certain applications but there is also an overhead of the order of $O(z^w)$ involving the storage of the prediction model, where z is the number of clusters in the *training* phase and w is the window size.

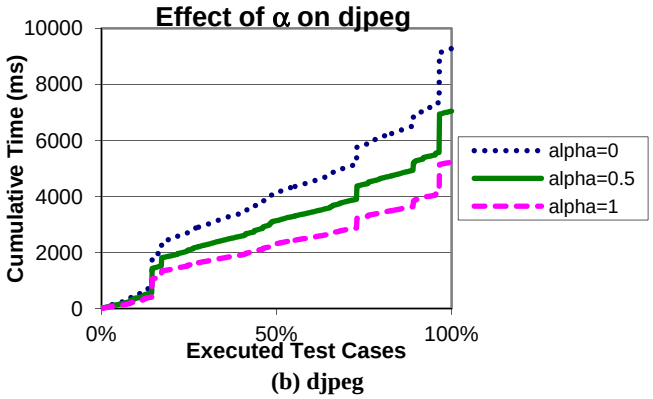
Table 5 shows the number of clusters generated by the EM algorithm and the number of distinct predictions for each application when 100% of the test cases are used for training. Although the maximum (and theoretical) number of predictions can be substantial, the actual number of predictions in practice ranges from 2.3% to 11.1% of the theoretical maximum. (We also note that it is possible to limit the number of clusters generated further by the EM algorithm.) Suppose that each prediction value on average takes 10 bytes, which include the storage required for the data and indexes. The amount of memory used is no more than 73 Kbytes.

Table 5. Clusters and Prediction Overhead

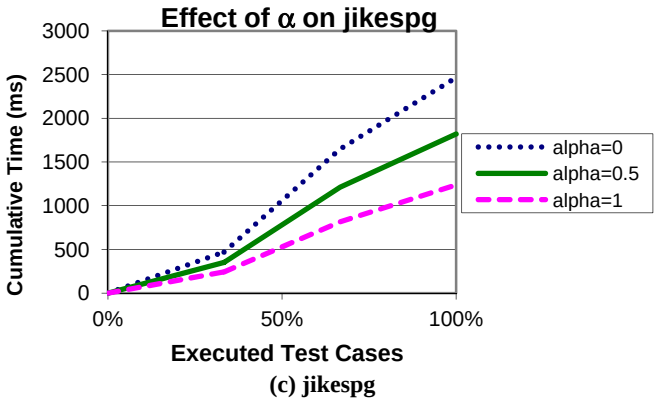
Application	$\|S\|$	z	Theoretical # of Predictions	Actual # of Predictions
minigzip	40940	16	$16^4 = 65536$	7300
djpeg	3920	8	$8^4 = 4096$	103
jikespg	2663	15	$15^4 = 50625$	1162



(a) minigzip



(b) djpeg



(c) jikespg

Figure 5. Effects of α

In summary, *EClass* can be implemented with little runtime and energy overheads, and allows a flexible tradeoff between performance and energy savings.

4.9 Threats to Validity

In this study, we experimented with C programs from the CSiBE benchmark with test cases obtained from the public domain. Despite our efforts in choosing applications with different I/O and computation intensity, and conducting realistic evaluations on real-life applications and test cases, these applications may not represent a full variety of possible program execution patterns. The experimental results may differ if the frequency assignment interval is changed, although many interval-based algorithms operate in fixed periods. It is always preferable to include more subjects in an experiment. To strike a balance between our effort and the scale of the experiment (which is already comparable to existing work [2][14] [21]), we settle for the scale reported in this paper.

Another factor that may affect our results is the platform-dependent parameters such as those in the energy model. For instance, if the off-chip energy consumption is unreasonably high on a hardware platform, the difference between DVFS techniques may become minimal because the on-chip energy consumption is relatively small. Our customization of XEEMU may also introduce faults into the simulator, even though we have spent our best effort to validate our experimental results and related published results.

We analyze the results through the energy savings achieved by individual techniques as reported by the simulator. Using other simulators or metrics such as the temperature of the hardware may produce different results.

5 Related Work

Previous research tried to reduce energy consumption of different system components such as memory and wireless networking components by exploiting the low-power modes or execution behavior of these components [18][25][27][33] [37]. In many computing systems, however, CPU is still the most important core component and the primary power consumer among components with multiple power operation modes. Previous research in cluster-wide power management also focuses on task scheduling over DVFS-capable processors [6].

Chen and Kuo [8] summarized numerous research results on DVS platforms for real-time systems in a survey paper. They categorized the techniques by the types of real-time jobs, scheduling, and processor architectures; the methodology to extract slack time for energy savings; the scope of energy consuming components considered and options available to reduce energy; the available information of workload characteristics; and the optimization objectives involved. The literature reviewed included analytical studies conducted earlier, optimal algorithms that run in exponential time, and heuristics algorithms that run in polynomial time with approximation bounds. Among the heuristics algorithms reviewed, none of them was reported to utilize machine learning as the primary technique. Thus, we believe that machine learning is a novel extension to the study of energy-efficient task executions.

Borrowing the terminology from software testing, existing approaches can be classified as "black-box" or "white-box". Black-box approaches rely on different runtime statistics to predict future execution behavior, and are oblivious to program structure. Although these techniques may not accurately predict changes in CPU utilization, they are usually praised for their ease of implementation and wide applicability. On the other hand, white-box approaches rely on the knowledge of program structure or implementation details. Program phases are identified using source code analysis or knowledge of execution behavior, and are used to assist frequency assignment. Despite these requirements, white-box techniques are usually more fine-grained and perform better for the corresponding tasks.

Recent research in the area of energy conservation of computing devices focus on high-level aspects such as optimizing computing resources in a cloud by consolidating the tasks onto the minimum number of servers [30]. Another recent track of publications tackles the problem in specific domains such as data-flow applications [2], real-time systems [39], and services with well-defined energy and delay cost functions [1]. Little breakthrough has been achieved for energy conservation of computing tasks in a more general context.

Energy-aware task scheduling in real-time embedded systems has also generated much attention in the past decade. Research in real-time systems usually incorporates other dimensions such as task completion deadlines [20] and reliability [38].

5.1 Black-Box Approaches

Classic examples of black-box frequency scheduling algorithms are the *PAST* algorithm proposed by Weiser et al. [35] and its closely-related variant AVG_N. The algorithms check at predetermined intervals the number of busy and idle CPU cycles, and assume that the CPU will behave similarly in the next interval. They differ in the sense that AVG_N computes the exponential moving average of the previous N intervals with decay, whereas *PAST* simply considers the current interval.

Grunwald et al. [16] evaluated these algorithms with three frequency switching policies: *one*, *double*, and *peg*. When utilization exceeds or falls below predefined thresholds, *one* increments or decrements frequency by one step; *double* tries to double or halve the frequency; and *peg* sets the frequency to the highest or lowest value. Experiments were conducted on four subject applications, but even the best heuristic policy (*PAST* + *peg*) could not achieve the best possible scheduling deduced manually by the authors.

Choi et al. [9] examined the average CPU cycles and memory cycles per instruction computed from event counters in the Performance Monitoring Unit (PMU) on the processor. The ratio is used to estimate the optimal frequency for an interval taking into account a user-defined performance loss

constraint. Similar to *PAST*, this technique relies on historical runtime statistics, but utilizes a better frequency switching heuristic than *PAST*.

Alimonda et al. [2] focused their attention on data-flow applications assuming the multiprocessor systems-on-chip (MPSoC) architecture. This architecture allows frequency and voltage selection in several cores. Assuming the knowledge of the queuing pipeline such as the throughput gain of the input and output queues, their approach monitors the buffer occupancy and makes frequency adjustments if it is below or above the desirable value in a nonlinear feedback loop.

In another recent study, Dhiman and Rosing [14] developed an interval-based technique based on the statistics collected from a PMU capable of monitoring four hardware events at a time. They found that the "number of instructions executed" (INST), "data cache misses" (DCACHE), "cycles instruction cache could not deliver instruction" (ICACHE), and "cycles processor is stalled due to data dependency" (STALL) to be indicative of task characteristics. Runtime statistics are passed to an online learning algorithm in which each working voltage / frequency pair is related to an "expert" with a probability factor influenced by the statistics. At every interval, the algorithm updates the factors and chooses the expert (and the related voltage/frequency) with the highest probability. While this technique also relies on historical data from earlier intervals, it uses different statistics counters and frequency assignment policies based on online learning. Our approach is similar in that we also make use of runtime statistics, but instead of directly predicting the near future, we compare all the executions in a training period to identify the past and the most probable future utilization pattern.

Liu et al. [23] considered energy-aware task scheduling for a special type of real-time embedded system with an energy harvesting module and an energy storage module. Their approach considers the Maximum Power Point (MPP) and overhead in charging and discharging the battery, and tries to minimize the system-wide energy cost and make better frequency assignments and charging decisions under different battery and workload conditions. Built on top of AS-DVFS task scheduling, their algorithm estimates the overall cost when the harvested energy is more than required according to a pre-determined energy model. Depending on the lowest cost, it will choose one of the following actions: store the additional harvested energy provided that the battery is not fully charged, speed up to the supported frequency immediately lower than the real-valued frequency (f') that utilizes the extra energy, or speed up to the supported frequency immediately higher than f'.

Rizvandi et al. [29] approached the problem of slack reclamation for real-time tasks analytically using an optimal combination of frequencies supported by the processor. They proved that optimal energy consumption can be achieved by at most two discrete, adjacent supported frequencies given that the relationship between frequency and energy is convex, and that power is positively correlated to voltage and frequency. Their algorithm first computes the ideal real-valued frequency (f_{ideal}) by using all available slacks based on the task deadline. If the relationship between frequency and energy is convex, the task is simply executed at the supported frequencies immediately lower and higher than f_{ideal} for computed durations. Otherwise, the supported set of frequencies is divided into two subsets: frequencies lower than f_{ideal} and those higher than f_{ideal}. The algorithm then searches for the two frequencies (one from each set) that combine to achieve the lowest energy consumption.

5.2 White-Box Approaches

Tiwari et al. [32] was the first to conduct power analysis on assembly code. Electrical current was measured when running each type of assembly instructions in infinite loops. Assembly programs were divided into basic blocks, where energy consumption was estimated per instruction type within each block. Block-estimates were multiplied by the number of times each block was executed, and penalty for cache misses was added into the final overall estimate. They also studied the circuit state overhead when executing a pair of instructions costs more than the sum of executing the same instructions individually. Although the analysis by Tiwari et al. focused on low-level assembly code, their framework has been extended to higher level software components for energy optimization.

Liu et al. [24] patched an existing Linux kernel to allow application programmers to adjust processor speed from inside the program according to specific needs of individual applications. They defined three application-independent steps (estimate demand, estimate availability, and determine speed) and utilized the power management architecture on various applications: an MPEG decoder, a videoconferencing tool, a word processor, a web browser, and a batch compilation task. Programmers can access and define a preprocess frequency schedule through new system calls introduced by the kernel patches and new modules. The kernel scheduler will set the processor frequency accordingly when a managed task becomes active upon context switching. This technique has high flexibility because it allows different types of applications to take advantage of their own power saving opportunities. However it involves program modification and recompilation, which require expert knowledge and perhaps retesting of the affected modules.

Another class of algorithms exploits the tradeoff between the quality of service/solution (QoS) and performance/energy by perturbing the computation. Baek and Chilimbi [4] developed a framework to improve performance and conserve energy for programs that can make use of approximations of expensive functions or loops such as the estimation of π. The framework requires that the programmer provide a function that computes QoS metrics given training function inputs or loop iteration counts. Approximation is carried out by running "light" versions of the expensive functions or fewer loop iterations based on QoS requirements and the computed metrics on QoS loss. To adapt to runtime performance variations, the framework also includes a recalibration mechanism to correct for deviations observed during executions. For programs suitable for controlled approximation, the performance improvements and energy savings of

this technique can be substantially greater than other techniques discussed, as executions can be terminated prematurely depending on the QoS requirements. Our approach is different from the white-box approaches described above. First, it is a black-box technique and is independent of the underlying programming language. Second, our approach does not alter execution paths or program output, nor does it require expert customization of program executions.

6 Conclusion and Future Work

Energy efficiency is an important issue in many areas of computing. In this paper, we have presented a framework for an energy-aware frequency assignment algorithm for DVFS processors. It distinguishes program phases during training executions and utilizes such information to make online frequency assignment decisions efficiently. We have presented an algorithm known as *EClass*. It samples intervals of executions and computes optimized frequencies for these intervals, followed by clustering the intervals into groups. During an online execution, *EClass* monitors the runtime statistics for the execution of an interval and uses a lookup approach to identify the best matched optimized frequency for the upcoming interval of the execution.

An *EClass* algorithm was implemented in Java and was evaluated against two existing algorithms on a cycle-accurate power simulator using three real-life applications. Results show that the proposed algorithm can, on average, save 9.1% more energy than another black-box algorithm based on online learning. Although our algorithm is general, the presence of a training phase requires some efforts prior to using the algorithm on the fly. An adapted algorithm *Target* is specifically customized for optimizing executions of the same test case.

A particular contribution of this paper is that according to experimental results, although classification can be much less efficient (and hence requiring more energy) than simply setting a value as in *Online*, the cost can be well compensated. The results show that energy-awareness research can benefit from advanced algorithms.

In the future, we will investigate whether the window size and checking interval can be automatically deduced and optimized for a task. Previous work [28] has shown that program phases are more pronounced with shorter intervals. Finer granularity may improve the result of our technique provided that the overhead associated with traditional DVFS can be overcome. We are also eager to extend the technique to multi-threaded, multi-core, and virtualized environments. One of the major challenges is to model the aggregated execution behavior of threads running simultaneously on cores that share common resources such as L2 cache. Recent research on execution time over multi-core systems (such as [31]) usually requires detailed analysis of the hardware to achieve accurate runtime predictions. While accuracy is important in areas such as Worst-Case Execution Time (WCET) analysis, approximations may be sufficient for the purpose of energy-efficiency. A promising approach is to extend the training phase of our technique to estimate the aggregated execution behavior of various tasks with different utilization patterns. We also believe that cross-platform training can be interesting, albeit challenging.

Our work and its experience have not considered the influences of power-saving features of existing operating systems and different hardware architectures. A future extension on this direction is useful. Moreover, the number of performance parameters provided by an operating system is much richer than what we have used in our experiment. It will be interesting to develop a generic methodology to select a good combination so that our work can be automatically tailored for in a specific computing environment.

References

[1] S. Albers. Energy-efficient algorithms. *Communications of the ACM*, 53 (5): 86–96, 2010.

[2] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini. A feedback-based approach to DVFS in data-flow applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28 (11): 1691–1704, 2009.

[3] H. Anton and C. Rorres. *Elementary Linear Algebra: Applications Version*. Wiley, New York, NY, 2010.

[4] W. Baek and T.M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2010*), pages 198–209. ACM Press, New York, NY, 2010.

[5] V. Berten, C.-J. Chang, and T.-W. Kuo. Managing imprecise worst case execution times on DVFS platforms. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (*RTCSA 2909*), pages 181–190. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[6] L. Bertini, J.C.B. Leite, and D. Mossé. Power optimization for dynamic configuration in heterogeneous web server clusters. *Journal of Systems and Software*, 83(4):585–598, 2010.

[7] T.D. Burd and R.W. Brodersen. Energy efficient CMOS microprocessor design. In *Proceedings of the 28th Hawaii International Conference on System Sciences* (*HICSS-28*), volume 1, pages 288–297. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[8] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (*RTCSA 2007*), pages 28–38. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[9] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Pro-*

ceedings of the Conference on Design, Automation and Test in Europe (*DATE 2004*), volume 1, pages 4–9. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[10] *CLOC: Count Lines of Code*. Available at http://cloc.sourceforge.net/. Last access May 2011.

[11] G. Contreras and M. Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design* (*ISLPED 2005*), pages 221–226. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[12] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B* (*Methodological*), 39 (1): 1–38, 1977.

[13] J.E. Dennis, Jr. and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics, Vol. 16. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1996.

[14] G. Dhiman and T.S. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design* (*ISLPED 2007*), pages 207–212. ACM Press, New York, NY, 2007.

[15] *GCC Code-Size Benchmark Environment* (*CSiBE*). Department of Software Engineering, University of Szeged. Available at http://www.inf.u-szeged.hu/csibe/. Last access April 2011.

[16] D. Grunwald, C.B. Morrey III, P. Levis, M. Neufeld, and K.I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation* (*OSDI 2000*), volume 4, article no. 6. USENIX Association, Berkeley, CA, 2000.

[17] Z. Herczeg, D. Schmidt, Á. Kiss, N. Wehn, and T. Gyimóthy. Energy simulation of embedded XScale systems with XEEMU. *Journal of Embedded Computing*, 3 (3): 209–219, 2009.

[18] H. Huang, P. Pillai, and K.G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of the USENIX Annual Technical Conference* (*ATEC 2003*), article no. 5. USENIX Association, Berkeley, CA, 2003.

[19] *Intel XScale Core Developer's Manual*. Intel Corporation, 2004. Available at http://www.intel.com/design/intelxscale /273473.htm.

[20] E.Y.Y. Kan, W.K. Chan, and T.H. Tse. Leveraging performance and power savings for embedded systems using multiple target deadlines. *The 1st International Workshop on Embedded System Software Development and Quality Assurance* (*WESQA 2010*), in Proceedings of the 10th International Conference on Quality Software (*QSIC 2010*), IEEE Computer Society Press, Los Alamitos, CA, pages 473–480, 2010.

[21] J. Kim, S. Yoo, and C.-M. Kyung. Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(1):110–123, 2011.

[22] C. Lin and S.A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium* (*RTSS 2005*), pages 410–421. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[23] S. Liu, J. Lu, Q. Wu, and Q. Qiu. Load-matching adaptive task scheduling for energy efficiency in energy harvesting real-time embedded systems. In *Proceedings of the ACM/IEEE International Symposium on Low-Power Electronics and Design* (*ISLPED 2010*), pages 325–330. IEEE Computer Society Press, Los Alamitos, CA, 2010.

[24] X. Liu, P. Shenoy, and M.D. Corner. Chameleon: application-level power management. *IEEE Transactions on Mobile Computing*, 7 (8): 995–1010, 2008.

[25] S. Misra, S.K. Dhurandher, M.S. Obaidat, P. Gupta, K. Verma, and P. Narula. An ant swarm-inspired energy-aware routing protocol for wireless ad-hoc networks. *Journal of Systems and Software*, 83(11):2188–2199, 2010.

[26] *Most Popular Websites on the Internet*. Available at http://mostpopularwebsites.net/. Last access April 2011.

[27] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI 2008*), pages 323–336. USENIX Association, Berkeley, CA, 2008.

[28] K.K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (*ISCA 2009*), pages 302–313. ACM PRESS, New York, NY, 2009.

[29] N.B. Rizvandi, J. Taheri, and A.Y. Zomaya. Some observations on optimal frequency selection in DVFS-based energy consumption minimization. *Journal of Parallel and Distributed Computing*, 2011. doi: 10.1016/j.jpdc.2011.01.004.

[30] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems* (*HotPower 2008*). USENIX Association, Berkeley, CA, 2008.

[31] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*

(*ISCA 2008*), pages 363–374. IEEE Computer Society Press, Los Alamitos, CA, 2008.

[32] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration Systems*, 2 (4): 437–445, 1994.

[33] M. Wang, Y. Wang, D. Liu, Z. Qin, and Z. Shao. Compiler-assisted leakage-aware loop scheduling for embedded VLIW DSP processors. *Journal of Systems and Software*, 83(5):772–785, 2010.

[34] *Wasabi Systems GNU Tools Version 031121 for Intel XScale Microarchitecture*. Intel Corporation, 2003. Available at http://www.intel.com/design/intelxscale/dev_tools/031121/wasabi_031121.htm. Last access April 2011.

[35] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (*OSDI 1994*), article no. 2. USENIX Association, Berkeley, CA, 1994.

[36] *Weka 3: Data Mining Software in Java*. The University of Waikato. Available at http://www.cs.waikato.ac.nz/ml/weka/. Last access April 2011.

[37] S.-L. Wu and S.-C. Chen. An energy-efficient MAC protocol with downlink traffic scheduling strategy in IEEE 802.11 infrastructure WLANs. *Journal of Systems and Software*, 84(6):1022–1031, 2011.

[38] B. Zhao, H. Aydin, and D. Zhu. Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In *Proceedings of the IEEE International Conference on Computer Design* (*ICCD 2008*), pages 633–639. IEEE Computer Society Press, Los Alamitos, CA, 2008.

[39] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD 2006*), pages 528–534. ACM Press, New York, NY, 2006.