6-2022

# Runtime Energy Savings Based on Machine Learning Models for Multicore Applications

Vaibhav Sundriyal

Masha Sosonkina

Scientific Research Publishing

Journal of Computer and Communications, 2022, 10, 63-80
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

# Runtime Energy Savings Based on Machine Learning Models for Multicore Applications

**Vaibhav Sundriyal, Masha Sosonkina**

Department of Computational Modeling and Simulation Engineering, Old Dominion University, Norfolk, USA
Email: vsundriy@odu.edu, msosonki@odu.edu

**How to cite this paper:** Sundriyal, V. and Sosonkina, M. (2022) Runtime Energy Savings Based on Machine Learning Models for Multicore Applications. *Journal of Computer and Communications*, **10**, 63-80.
https://doi.org/10.4236/jcc.2022.106006

**Received:** March 29, 2022
**Accepted:** June 27, 2022
**Published:** June 30, 2022



## Abstract

To improve the power consumption of parallel applications at the runtime, modern processors provide frequency scaling and power limiting capabilities. In this work, a runtime strategy is proposed to maximize energy savings under a given performance degradation. Machine learning techniques were utilized to develop performance models which would provide accurate performance prediction with change in operating core-uncore frequency. Experiments, performed on a node (28 cores) of a modern computing platform showed significant energy savings of as much as 26% with performance degradation of as low as 5% under the proposed strategy compared with the execution in the unlimited power case.

## Keywords

Machine Learning, RAPL, DVFS, Uncore Frequency Scaling, Energy Savings, Performance Modeling

## 1. Introduction

Modern computing systems are being increasingly controlled by their power consumption ranging from node components to a full fledged data center. The power/energy constraints are due to manifold reasons with technical and economical costs being the primary. To reach the exascale, modern computers must still nearly double their performance while the further increase in power consumption becomes prohibitive. Therefore, power/energy consumption becomes a major obstacle to application scalability, availability, and affordability, and it is urgent to develop techniques that optimize energy consumption while maximizing performance.

On the other hand, such optimization is a difficult task due in large part to a 1)

great variability in modern high-performance application workloads, and 2) complexity of modern hardware architectures. These two factors have to be accurately modeled to predict runtime performance under different power levels. Existing analytical and heuristic models fall short of this task because they cannot account for the multitude of hardware characteristics as they relate to the application dynamic changes. Recently, machine learning (ML) has been proposed as an effective alternative to modeling application time-to-solution under different dynamic voltage and frequency scaling (DVFS) [1] and uncore frequency scaling (UFS) levels. Note that the *uncore* encompasses those processor functions that are not handled by the core, such as L3 cache and on-chip interconnect. In this work, machine learning models are first investigated for their use during the runtime performance modeling of a diverse set of application workloads exhibiting dynamically changing compute- and memory-intensities. The models are incorporated into a novel runtime strategy along with power and processor-frequency selections. The strategy aims to maximize energy savings under a user provided performance constraint. The strategy operates in a manner transparent to the application and utilizes a timeslice based approach to select appropriate frequencies for the next timeslice. In a nutshell, as main contributions, this work

- Investigated and applied ML models for predicting performance during the runtime:
  - Considered uncore frequency scaling as an independent variable.
  - Employed dimensionality reduction to obtain a set of predictor variables from performance events.
  - Collected data for prediction on scientific workloads with diverse memory-accesses and computational patterns, including a large-scale quantum chemistry package GAMESS [2].
  - Evaluated three different ML algorithms as to their prediction accuracy vs time.
  - Investigated and tested two types of ML model construction, *fully runtime* and *pretrained statically*.
- Proposed a transparent runtime strategy that incorporates the developed ML performance models to maximize energy savings.
- Compared the proposed runtime strategy with its counterpart developed in authors' prior work [3].

The rest of the paper is organized as follows. Section 2 provides the related work. Section 3 outlines a previously developed performance model, used here for comparisons. Section 4, first, discusses hardware performance-event data collection along with dimensionality reduction; then, evaluates several ML algorithms for their usage at the runtime and proposes two types of the ML model construction. Section 5 details the runtime strategy along with its implementation steps. Section 6 shows experimental results and comparisons with the prior approach. Section 7 concludes the paper.

## 2. Related Work

There have been many previous research efforts that propose to use machine learning strategies for power and energy savings in modern computing systems. They target both single- and multicore systems using supervised and reinforcement learning for power management, temperature management, and performance maximization under a power constraint.

In [4], a dynamic power management (DPM) technique is proposed for an arbitrary number of sleep states that shuts down idle components based on clustering of idle periods. In [5], authors have proposed a strategy for an arbitrary number of sleep states that minimize the power consumption under a given performance constraint. Wang *et al.* [6] present an online hierarchical mechanism with application-level scheduling for an embedded system minimizes the total power consumption and finds an optimal point for power-delay relation for the connected devices. Albeit [4] [5] [6] operate on a single-core platform, they are relevant to the current work because they also deal with transparent strategies to manage dynamically power consumption of the processor.

Similarly to the current work, the work in [7] uses DVFS along with a learning based prediction. In particular, it proposes a supervised-learning based power management framework for minimizing energy consumption on a multicore chip equipped with DVFS on each core. A Bayesian classifier is employed for predicting performance of each core per incoming task by observing a set of input features. This predicted state is further used to find an optimal power management action in a pre-computed lookup table.

Bartolini *et al.* [8] propose a distributed thermal management technique utilizing model predictive control and self-calibration for minimizing energy consumption under performance and temperature constraints. Specifically, each core takes a value of the predicted cycles per instruction (CPI) of the running task as input and chooses the minimum frequency value while satisfying the performance constraints. The work in [8] considers the power management with thermal and performance constraints and utilizes analytical modeling by just using the CPI performance event. The results are demonstrated in simulation only rather than in real time.

In [9], a power management and task allocation framework based on Q-learning is proposed to attain a trade-off between performance and power consumption while simultaneously following the temperature constraints. A reinforcement-learning based strategy is proposed in [10] to prepare a scheduling policy where system invokes DVFS and any penalty is considered in decision making through the learning process. The work in [11] proposes an online thermal management strategy for dual goal of maximizing performance and reducing thermal cycles under a temperature constraint. The policies are directed by a reinforcement learning algorithm and apply a power management to optimize the desired goal.

Using constrained energy minimization, modeling with the CPI performance

metric, and assessing the performance penalty due to DVFS are the aspects of [8] [9] [10] [11] that are most synergistic with the current work. While all the aforementioned research makes use of the machine learning paradigm to approach power management, none attempts to model directly the out-of-order (OOO) processor pipeline and to consider the uncore frequency scaling, which are important factors in gaining maximum energy savings for highly changeable workloads as shown in the current work on a multicore platform.

## 3. Overview of the Analytical Performance Model

In the authors' prior work [3], a frequency scaling runtime strategy that targeted both core and uncore power domains was proposed to save energy in parallel applications with a minimal performance loss. This strategy relied on the analytical performance and power models that were also developed by the authors. In particular, performance modeling of the core—uncore domain was expressed in the following Equation (1).

Assume $n$ levels of the core frequency and $m$ levels of the uncore frequency denoted $f_c(i)\quad i=1,\cdots,n$ and $f_u(j)\quad j=1,\cdots,m$, respectively, on a given processor. The effect of the core and uncore frequency on the micro-operations retired is identified by

$$f_c(i)=\mu\tau(i,j)\left(\mathrm{CPM}_{\mathrm{exe}}+\frac{f_c(i)}{f_c(1)}\left(\mathrm{LLC\_MISSES}\times\alpha\times\beta_j\right)\right), \tag{1}$$

where

- $\mu\tau(i,j)$ is the number of micro-operations retired per second at core frequency $f_c(i)$ and uncore frequency $f_u(j)$.
- $\mathrm{CPM}_{\mathrm{exe}}$ is the number of cycles per micro-operation retired.
- $\alpha$ ($0\le\alpha\le1$) is the processor out-of-order (OOO) overlap factor, which was determined experimentally.
- LLC_MISSES is the number of memory accesses per micro-operation retired in a second.
- $\beta_j$ is the number of cycles corresponding to the memory access latency at the uncore frequency $f_u(j)$.

While the strategy based on this model delivered promising results of 15.3% in energy savings with 5.3% performance loss, its shortcomings were observed in applications with memory-intensive workloads, such as iterative linear system solvers, for which the the strategy saved less than 10% of energy. Such shortcomings may be explained by rather crude estimates of memory accesses per micro-instruction, which were modeled by only one parameter, last-level-cache (LLC) misses in Equation (1). Adding more performance events to model complex application behavior appeared not feasible in the analytical expression because the interdependence of multiple events cannot be determined for the broadly applicable model. The single heuristic used in the model, the experimentally determined OOO factor, already showed its narrow applicability scope since it had to be tuned beforehand for a given processor.

To overcome these deficiencies of the analytical model, machine learning approaches are considered in this paper in combination with analytical modeling of power and frequency levels. By definition, machine learning is suitable to consider numerous parameters-features, such as multiple performance events here, for training and producing models that work without expressing the parameter dependencies explicitly.

## 4. Machine Learning Model Construction

In this section, first the relevant performance data to train and test the model is selected using the reasoning based on the processor operation and the nature of workloads. Then three different ML algorithms are evaluated for the time vs accuracy trade-off along with the investigation of two possible modes to perform ML training stage: during the runtime and pretraining statically.

### 4.1. Performance-Event Data Collection

In general, the workload behavior of an application varies throughout its execution, exhibiting memory- or compute-intensive patterns on a fine-grained scale. Hence, runtime performance modeling is typically done in small time intervals, called here *timeslices*. For modeling energy consumption, timeslices of a fixed duration on the order of the frequency scaling overhead have proven to be a good choice in the authors' earlier work (see e.g., [12]), where it has been shown that the timeslices of 250 ms incur a low modeling overhead and are sustainable for large-scale applications, such as GAMESS quantum chemistry calculations, which are considered in the present work as well.

#### 4.1.1. GAMESS Overview

GAMESS is one of the most representative freely available quantum chemistry applications used worldwide to do *ab initio* electronic structure calculations. A wide range of quantum chemistry computations may be accomplished using GAMESS, ranging from basic Hartree-Fock and Density Functional Theory computations to high-accuracy multi-reference and coupled-cluster computations.

The central task of quantum chemistry is to find an (approximate) solution of the Schrödinger equation for a given molecular system. An approximate (*uncorrelated*) solution is initially found using the Hartree-Fock (HF) method via an iterative *self-consistent field* (SCF) approach or restricted HF (RHF), and then improved by various *electron-correlated* methods, such as second-order Mø ller-Plesset perturbation theory (MP2). The SCF-HF and MP2 methods are implemented in two forms, namely *direct* and *conventional*, which differ in the handling of electron repulsion integrals (ERI, also known as *2-electron integrals*). Specifically, in the conventional mode all ERIs are calculated once at the beginning of the interactions and stored on disk for subsequent reuse whereas in the direct mode ERIs are recalculated for each iteration as necessary. The SCF-HF iterations and the subsequent MP2 correction find the energy of the molecular

system, followed by evaluation of energy gradients.

GAMESS mesoporous silica nanoparticles (MSN) inputs were used for the ML model training and testing. Specifically, MSN 11-, 16-, 22-, and 32-fragment RHF calculations using a state-of-the-art effective fragment molecular orbital (EFMO) method were considered. The inputs are referred to as msn-11, msn-16, msn-22, and msn-32 in the rest of the paper.

Additionally, several NAS parallel benchmarks (NPB) [13] were chosen to further increase the mix of compute- and memory-intensive workloads with common scientific irregular computation patterns. Both NPB and GAMESS were executed on the Xeon based platform and data on certain performance events—as detailed below—was collected for a 250 ms timeslice duration. The core and uncore frequency ranges on the Xeon platform are 1.2 - 2.3 GHz and 1.4 - 2.7 GHz, respectively. Instead of collecting event data for each and every (core, uncore) frequency pair, only four bracketing combinations were considered—(2.3, 2.7), (1.2, 2.7), (2.3, 1.4), and (1.2, 1.4)—to ensure that, at prediction time, no extrapolation is necessary and an interpolation is sufficient.

#### 4.1.2. Selection of Performance-Counter Events

The hardware platform used in this work employs an Intel Xeon E5-2695 v3 processor that comprises 14 cores. Each processor core is equipped with multiple hardware performance counters, which provide runtime count for such events as micro-operations retired and L3 cache misses. Similar to the analytical model in Equation (1), the ML performance model considers the micro-operations retired as the measure of processor performance at different core—uncore frequencies to predict performance in a given timeslice.

The Xeon E5-2695 v3 processor has approximately 190 performance events[1], which makes it intractable to read all of them and use in an ML model for the runtime performance prediction. Therefore, only certain most relevant and impactful, events must be selected. In particular, the following procedure was undertaken. All the events, along with their event codes, are scraped and those events that are a part of aggregation or are not related to processor performance have been omitted. For example, there are about twelve performance events that deal with resource stalls (e.g., RESOURCE_STALLS.LB, RESOURCE_STALLS.ROB, RESOURCE_STALLS.RS), eleven of which are aggregated in a single event RESOURCE_STALLS.ANY that is taken as the one dealing with stalls in the ML model. By continuing with agrregation in other event groups, the number of events was brought down to 45, thereby reducing the data-variable dimensionality by more than 75%.

A custom C language program had to be developed to periodically and selectively collect the performance-event data as follows. To read a specific event, the event code is first written to a given performance-event select register whose address starts from $0 \times 186$. Then the value of the event is read from the corresponding performance-event counter whose address starts from $0 \times C1$. Next, the

[1]https://perfmon-events.intel.com/snbep.html.

event values were monitored at the runtime. It was observed that many event values were at zero. Therefore, all such events were also dropped, which left only ten events further reducing the dimensionality drastically.

Table 1 shows the resulting set of events considered in this work along with their ranges on the NPB and GAMESS inputs. Note that, for the input to ML models, the event values are to be normalized by the UOPS_RETIRED event value of the corresponding timeslice, so that the events are made invariant to core—uncore frequency changes in order to increase the accuracy of predictions at the runtime and to further reduce the number of events to nine.

### 4.2. Evaluation of Different ML Algorithms

Execution performance modeling may be treated as the multiple regression problem. Hence, three appropriate algorithms tackling this problem were chosen as follows: Linear Regression (LR), K-Nearest Neighbors (KNN), and Random Forest Regressor (RFR) [14]. The train-test validation was used for the evaluation on the collected data (Section 4.1). Due to a large variety of workloads in the dataset, the training data may be assumed to come from many different distributions, and thereby allowing the model to better generalize on a variety of test data. Furthermore, the stratified sampling [15] was employed to avoid random sampling bias in the dataset.

In the authors' previous works [3] [12], it was determined that the LLC miss count (see event #1 in Table 1) was considered a greatly important parameter for modeling because an LLC miss leads to a DRAM access during which there is an opportunity to reduce the processor frequency and, thereby, obtain energy savings with minimum performance penalty. As a consequence, the train and

Table 1. Resulting set of performance events considered as independent variables with corresponding ranges in micro-operations.

| # | Event Name | Description | Range, ($\mu$ops) |
|---|---|---|---|
| 0 | UOPS_RETIRED | counter for $\mu$ops retired | 9.159146e7 - 8.133418e9 |
| 1 | MEM_LOAD_UOPS_RETIRED.LLC_MISS | miss in last-level L3 cache (LLC) | 1.0e−6 - 1.7e−2 |
| 2 | BR_INST_RETIRED.ALL_BRANCHES | all (macro) branch instructions retired | 3.5e−3 - 2.5e−1 |
| 3 | BR_MISP_RETIRED.ALL_BRANCHES | all mispredicted macro branch instructions retired | 3.0e−6 - 6.1e−3 |
| 4 | LONGEST_LAT_CACHE.MISS | core-originated cacheable demand requests that missed LLC | 2.0e−6 - 3.1e−2 |
| 5 | LONGEST_LAT_CACHE.REFERENCE | core-originated cacheable demand requests that refer to LLC | 2.4e−6 - 1.0e−1 |
| 6 | MEM_UOPS_RETIRED.ALL_LOADS | counter for load $\mu$ops retired | 2.8e−4 - 2.14e0 |
| 7 | MEM_UOPS_RETIRED.ALL_STORES | counter for store $\mu$ops retired | 1.3e−3 - 1.1e0 |
| 8 | OFFCORE_REQUESTS.ALL_DATA_RD | demand and prefetch data reads | 1.3e−5 - 1.4e0 |
| 9 | RESOURCE_STALLS.ANY | resource-related stall cycles | 1.0e−5 - 5.6e−2 |

test sets have to be representative of a variety of LLC miss values in the entire dataset. Table 2 shows the ranges of LLC misses and the corresponding bin numbers into which the misses were sampled. Figure 1 presents the resultant training dataset distribution into bins based on the LLC misses incurred. Note that the relatively low and high values of LLC misses correspond to the compute- and memory-intensive application inputs, respectively. It has been observed that the test set exhibited a distribution similar to that shown in Figure 1. Hence, the sampling categories in Table 2 were chosen for use in the ML algorithms considered here.

After dividing the entire data set into train (80%) and test set (20%), which are the commonly used ratios for evaluating machine learning models [14], the prediction accuracy of the three algorithms was determined as shown in Figure 2. It can be observed from Figure 2 that, while the KNN and RFR algorithms have much higher train accuracy than LR does so, they all have nearly the same test accuracy. A large discrepancy between train and test accuracies of KNN and RFR is intrinsic to their design (see, e.g., [14]), which is different from that of LR.

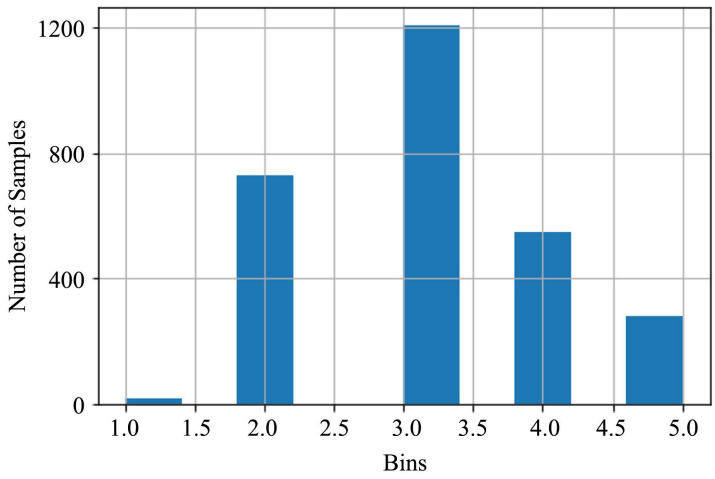


Figure 1. Assignment of the train data set samples to bins based on their LLC-misses count.

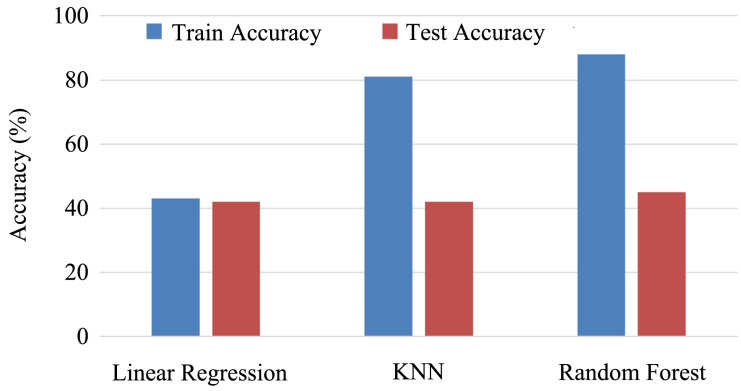


Figure 2. Train and test accuracy for the three ML algorithms.

**Table 2.** Ranges of LLC misses and their associated bins.

| Range of LLC Misses | Bin |
|---|---|
| 0.00 - 0.05 | 1 |
| 0.05 - 0.1 | 2 |
| 0.1 - 0.2 | 3 |
| 0.2 - 0.3 | 4 |
| 0.3 - 1.0 | 5 |

In this work, an important ML algorithm selection criterion is the time to use the obtained ML model dynamically. In particular, the said time has to be on the order of the timeslice duration considered here so that the overall application performance is not degraded. **Figure 3** shows the time to predict a single data point, *i.e.*, to apply the ML model only once to predict the runtime performance, spent by the three algorithms. It can be seen that this time is the lowest for LR. Additionally, LR is designed to accurately capture a relationship between the response and predictor variables, contrary to the other two ML algorithms that predict based on the boundary estimates. Therefore, LR is selected to model performance in the runtime strategy developed in this work (see Section 5).

### 4.3. Training Stage: Performed Statically or Dynamically

When considering dynamic usage of the ML models, the prediction stage has to be always done at the runtime to react to the actual input into the model, which is changing on the timeslice scale in this work. Now, the training stage may be performed either *statically*, before the execution, resulting in the pretrained model applied in each timeslice during the execution, or *dynamically*, such that both (re)training and prediction are done in each timeslice on the newly acquired data as a part of the runtime strategy. Obviously, the overhead from application of the former—termed here *pretrained statically* or trn-Static for short—affects the performance less during each timeslice. However, trn-Static may lead to less accurate predictions, thereby affecting both the overall energy savings and time-to-solution. Furthermore, the OOO overlap factor $\alpha$ (see Equation (1)) is still determined experimentally, as in the analytical modeling, when trn-Static is used since all the model training is done completely offline similar to analytical modeling. The latter—termed here *fully runtime* or all-Runtime for short—appears the most agile and incorporates $\alpha$ implicitly in the model at each timeslice. The application of the ML model with all-Runtime too, if care is not taken, may come at a price of lesser accuracy and higher performance loss accumulating for the entire execution. To train with all-Runtime efficiently, both the training time and the training sample size have to be considered. **Figure 4** shows the time to train the model on the entire collected data set for the three algorithms. It can be observed from **Figure 4** that LR has the lowest time among the three algorithms, and thus, corroborates its selection in the runtime strategy. Also, **Figure 5** shows that LR exhibits a stable performance when the sample size

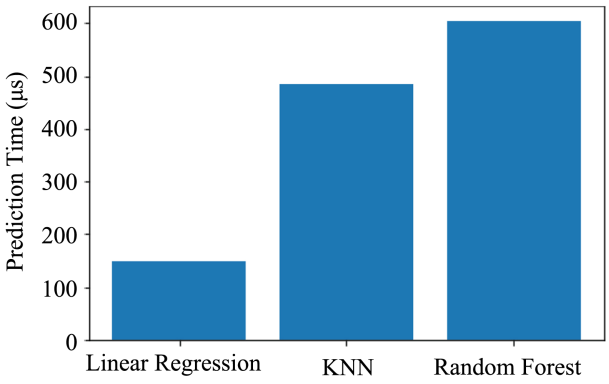


**Figure 3.** One-time application of the ML model.

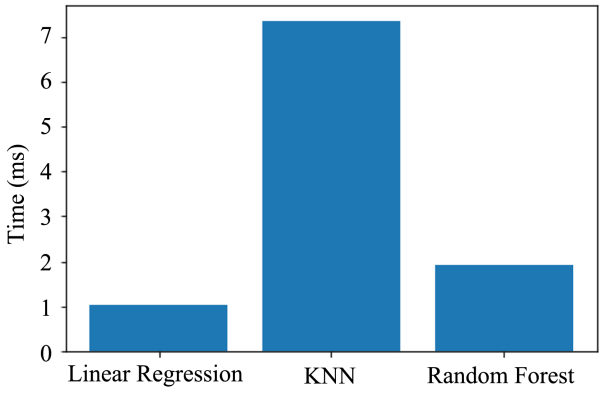


**Figure 4.** Model timings on the entire dataset.

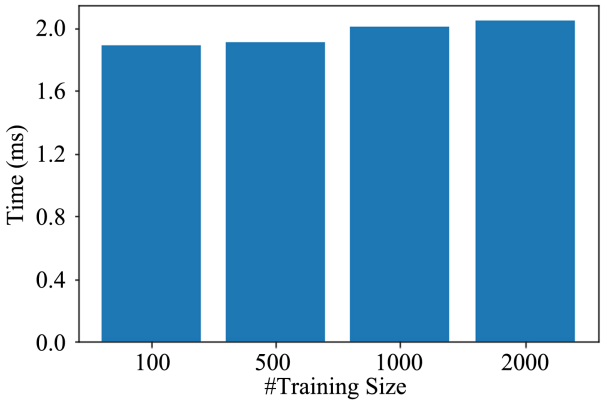


**Figure 5.** The LR model training on the increasing sample sizes.

grows from 100 to 2000, which is another argument in support of the selection of LR.

## 5. Design and Implementation of the Runtime Strategy

This section outlines the combination of the proposed ML model with the ones for power and frequency level, which are similar to those used in authors' earlier work [3]. Then, it describes in detail the algorithmic steps implementing the strategy. The proposed strategy utilizes ML modeling to maximize energy sav-

ings of a parallel application on a compute node. In each timeslice, for each core, the strategy gathers the relevant performance-counter information, which is used to model the performance and power for the next timeslice to be executed in order to determine the next optimal core and uncore frequencies for each core, followed by the application of the chosen frequency pair before commencing the next timeslice.

To better predict the next timeslice performance characteristics, a few past neighboring timeslices may weigh in their actual characteristics along with the current predicted values. In particular, following the work in [3], a *history-window* predictor has been built into the strategy such that some function $g$ accepts a window of several neighboring values and outputs a prediction for the next timeslice. After each prediction, the window slides forward by one position.

The performance loss tolerated due to energy savings has to be bounded (typically, at no more than 10%). Hence, the potential performance loss must be calculated and kept within the upper bound for the core—uncore frequency prediction to be feasible. A frequency-pair subset $F$ contains all such feasible frequency pairs. The following expression calculates the performance loss $\delta\left(f_c(i), f_u(j)\right)$ as proposed in [16] when the application is executed on a core frequency $f_c(i)$ and uncore frequency $f_u(j)$ as compared with the execution at the highest, level 1, core and uncore frequencies.

$$\delta\left(f_c(i), f_u(j)\right) = \frac{\mu\tau(1,1) - \mu\tau(i,j)}{\mu\tau(1,1)}. \tag{2}$$

Note that the value of the micro-operations retired $\mu\tau$, relates directly to application performance.

### 5.1. Power Modeling

To account for the instantaneous power consumption in the proposed runtime strategy and to select the core and uncore frequencies that minimize the system energy under a performance constraint, the Intel RAPL tool [17] is used, which provides instantaneous processor power consumption. The processor power consumption, denoted $P_p(i,j)$ at the core and uncore frequencies $f_c(i)$ and $f_u(j)$, respectively, varies proportionally to the cube of the frequency values, as shown in [3]. Consequently, the value $P_p(i,j)$ may be expressed as

$$P_p(i,j) = k_1 \times f_c(i)^3 + k_2 \times f_u(j)^3, \tag{3}$$

where $k_1$ and $k_2$ are constants. The values of $k_1$ and $k_2$ were determined through a regression analysis similar to the one proposed in [3] Then, the total power consumption $P_T(i,j)$ of core, uncore, and DRAM domains is as follows:

$$P_T(i,j) = P_p(i,j) + P_m + P_{\text{static}}, \tag{4}$$

where $P_m$ is the memory power consumption (determined at the runtime from RAPL), and $P_{\text{static}}$ is the static power consumption of the three domains, determined to be 40 Watts using RAPL.

### 5.2. Choosing the Optimal Frequency Levels

Finally, given the predictions for the next timeslice $r$ of the performance and total power at all the available core—uncore frequency level pairs $(i, j), i = 1, \cdots, N$ and $j = 1, \cdots, M$, an optimal pair $\left(f_c(o_c), f_u(o_u)\right)$ has to be selected to minimize the performance loss (in Equationg (2)). In other words, a total energy minimization problem may be solved as follows:

$$P_T(o_c, o_u) \times \tau\left(1 + \delta(o_c, o_u)\right) = \min_{(f_c(i), f_u(j)) \in F} \left[ P_T(i, j) \times \tau\left(1 + \delta(i, j)\right) \right], \tag{5}$$

where $\tau$ is the fixed timeslice duration, the term $\tau\left(1 + \delta(i, j)\right)$ represents the next interval execution time, which is possibly larger than $\tau$ by factoring in the performance loss corresponding to the operation at a frequency from the feasible subset *F*.

### 5.3. Runtime Energy-Saving Algorithm

Figure 6 displays the steps of the algorithm underlying the proposed runtime strategy. Step 1 profiles the application for duration $\tau$ and obtains the relevant event values from the performance counters. Next, Step 2 predicts value of events for the next timeslice $r$ to be executed by using the *history-window* algorithm with the window of size of three, in which averaging as the $g$ function proved sufficient for the given workloads and timeslice duration. Step 3 calls either *fully runtime* or *pretrained statically* ML model with the LR algorithm (in function useML_LR) to predict the micro-operations retired for all the core—uncore frequency pairs, returned as set $M_\tau$.

Next (Step 4), a subset $F \in M_\tau$ is determined consisting of all those core—uncore frequency pairs for which the predicted performance loss does not exceed the performance-loss constraint $\gamma$. The threshold value of $\gamma$ is provided by the user while the actual resulting performance loss is measured using the

**Input Parameters:**
$\tau$ : Duration of the timeslice $r$.
$V$ : Number of timeslices.
$f_c(1), \ldots, f_c(N)$ : Available core frequencies ($N$).
$f_u(1), \ldots, f_u(M)$ : Available uncore frequencies ($M$).
$\gamma$ : User-defined performance loss bound.
*ml-type* : `trn-Static` or `all-Runtime`.
**Algorithm:**
**Step 1.** Execute application in timeslice $r = 1$ and gather performance-counter event information.

**For** ($r = 2$, $r \leq V$, $r++$) **do**
**Step 2.** Predict the event value in $r$ using the *history-window* predictor.

**Step 3.** $\mathcal{M}_\tau$ = `useML_LR`(*ml-type*), where $\mathcal{M}_\tau = \{\mu\tau(i, j) \mid i = 1, \ldots, N \text{ and } j = 1, \ldots, M\}$.

**Step 4.** Determine feasible subset $F$ of core–uncore frequency pairs with $\delta \leq \gamma$ in Eq. (2).

**Step 5.** Calculate $P_T(i, j)$ for all $i = 1, \ldots, N$ and $j = 1, \ldots, M$ from Eq. (4)

**Step 6.** Choose the operating core $f_c(o_c)$ and uncore $f_u(o_u)$ frequency in timeslice $r$ as in Eq. (5).

**Step 7.** Execute application for the duration $\tau$ at ($f_c(o_c)$, $f_u(o_u)$) in $r$.

**EndFor**

**Figure 6.** Pseudo-code for the ML-based energy-saving runtime strategy.

number of micro-operations retired at the end of a timeslice. In Step 5, the power consumption for all the core—uncore frequency combinations is obtained. Then, in Step 6, an appropriate operating frequency pair is chosen from solving the energy minimization problem as described in Section 5.2.

## 6. Experimental Results

The experiments were performed on a compute node having two Intel Xeon E5-2695 v3 14 core Haswell-EP processors with 32 GB (4 × 8 GB) of DDR4. The core and uncore frequency ranges are 1.2 - 2.3 GHz and 1.0 - 2.6 GHz, respectively. To measure the socket and DRAM power, Intel RAPL API was used. The user-defined performance-loss tolerance $\gamma$ was taken as 10%, which is a typical value to allow for energy savings (see, e.g., [18]).

### Performance and Energy Savings

Figure 7 shows the performance degradation of the proposed runtime strategy relative to the performance with both the core and uncore frequency levels staying at their maximum. The four NAS benchmarks are shown as "xx.yy.zz", where "xx", "yy", and "zz" denote benchmark name, class, and number of processes used, respectively. The four GAMESS MSN inputs are distinguished by their fragment sizes (11, 16, 22, and 32). The proposed runtime strategy is also compared with the one developed earlier in [3]—termed here eq-PerfMod—the performance model of which is outlined in Section 3.

The EP benchmark is invariably CPU-intensive throughout the execution with its performance degrading in a linear manner with the reduction in the core frequency. Therefore, the all-Runtime and eq-PerfMod execute EP at the lowest

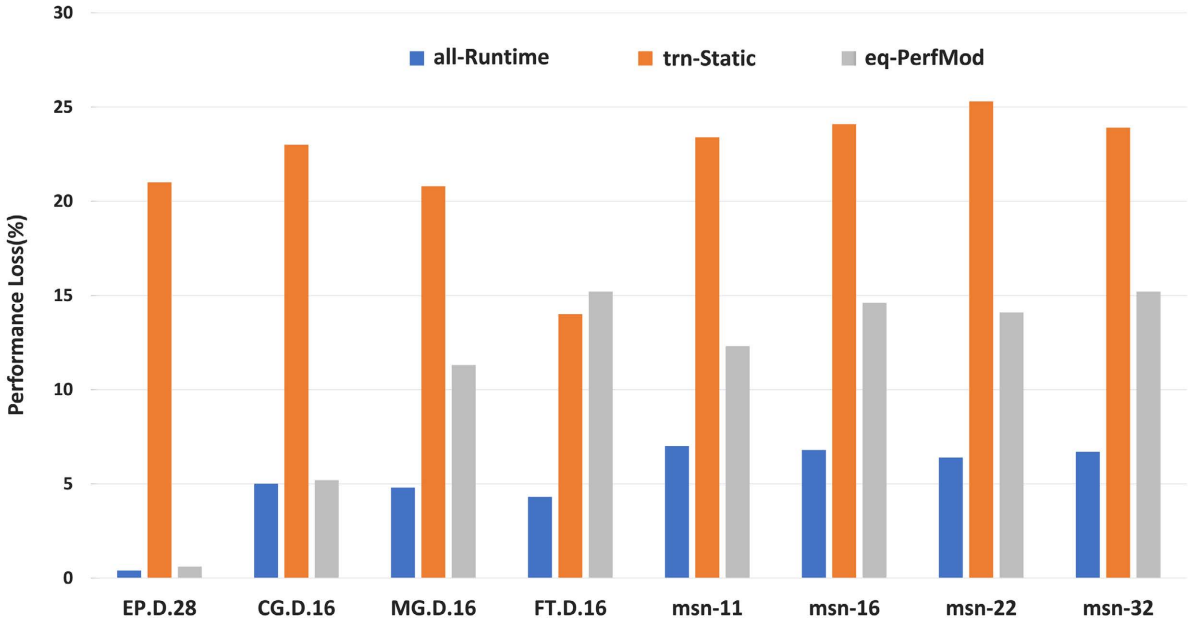


Figure 7. Performance loss for the NAS and GAMESS inputs when operated under the proposed runtime strategy on a 28-core Haswell-EP node.

uncore frequency all the time. On the other hand, trn-Static executes the EP benchmark primarily at the 1.6 GHz core frequency and the 1.8 GHz uncore frequency, thereby significantly degrading its performance. The reason for trn-Static to select such a low core frequency for EP is that it has been *pretrained* on data, which came from a broad set distributions, while EP is showing an obvious compute-intensive penchant. Similarly, a poor prediction tendency of trn-Static may be observed for the memory-intensive inputs, such as the CG benchmark. Although the value of the LLC misses for CG is much higher than that for EP, the high bandwidth DDR4 is able to significantly overlap computational work with memory accesses [19]. Hence, its memory intensity is not enough to warrant a significant reduction in the core frequency, which is correctly detected by both trn-Static and eq-PerfMod. For the memory-intensive MG, trn-Static reduces both core and uncore frequency to 1.5 GHz and experiences significant performance degradation of ~20%. The reason for trn-Static performing poorly is that it fails to interpolate and generalize on the data generated during the runtime. The ML model with trn-Static seems to average when generalizing on the unseen data. Therefore, trn-Static performs poorly for all the NAS benchmarks. The ML model with all-Runtime, on the other hand, for a given workload, adds training on a single input of the currently executed application per timeslice and, thus, is able to generalize much better. Such an advantage of all-Runtime, is even more pronounced in the MSN inputs.

For the four MSN inputs, only all-Runtime succeeds in maintaining the performance constraint by primarily executing he MSN inputs at 1.1 GHz uncore frequency and the highest core frequency. The dynamic changes in the workload parameters listed in **Table 1** are even more pronounced for the MSN inputs than those are for the NAS benchmarks. Hence, the poor performance of the trn-Static and eq-PerfMod is observed, which is due to the same reasoning as for the NAS benchmarks. They both operate the MSN inputs with the core frequency between 1.4 and 1.6 GHz and the uncore frequency between 1.5 and 1.7 GHz. Although the two switch to similar core—uncore frequency ranges, the trn-Static tends to execute the MSN inputs at the lower end of the range of the core frequencies. Overall, across all the eight inputs, the average performance loss incurred by all-Runtime, trn-Static, and eq-PerfMod was 5.1%, 21.8% and 10.9%, respectively.

The eq-PerfMod, despite depending on the heuristic performance analysis, is dynamic in nature, contrary to trn-Static, and is updated with the most current LLC misses in each timeslice, thereby adapting at the runtime and yielding better energy savings than those obtained by the trn-Static in all the tested applications. **Figure 8** shows the energy savings corresponding to the performance losses in **Figure 7**.

Since both all-Runtime and eq-PerfMod operated the EP benchmark at the lowest uncore frequency, they were able to reduce energy consumption by more than 20%. The trn-Static on the other hand, provided only 13% of energy savings. A maximum of only 4.8% in energy savings was achieved for CG benchmark

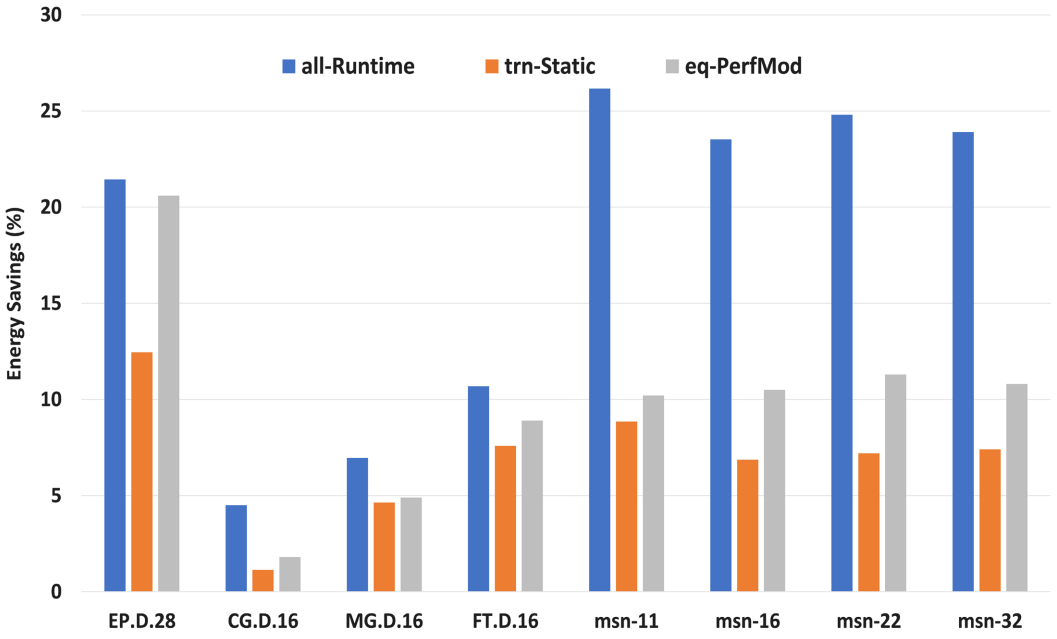


**Figure 8.** Energy savings for the NAS and GAMESS inputs when operated under the proposed runtime strategy on a 28-core Haswell-EP node.

since, overall, it is neither memory- nor compute-intensive due to the DDDR4 memory effects. For the MG and FT benchmarks, all-Runtime yields more energy savings than the other two strategy variants do and it saves 6.8% and 10.4% of energy, respectively. Note that, for CG, MG, and FT, the trn-Static and eq-PerfMod provide similar energy savings. However, trn-Static achieves them by aggressively applying frequency scaling, and thereby breaching the performance constraint while eq-PerfMod applies frequency scaling more carefully and yields a much smaller performance degradation. By comparing broadly, a minor trend is observed where trn-Static gradually starts to improve its prediction (cf. **Figure 8**) for the MSN inputs with the increase in the size of the MSN calculation. Larger MSN inputs afford more opportunities, in terms of the number of timeslices, for the trn-Static to pretrain on the stabilized workload parameters beyond the erratic initialization phase, which may skew smaller MSN calculations. Overall, smaller calculations lead to more sensitivity in the trn-Static due to fewer data points available to capture a given workload changes reliably. Therefore, longer execution traces are preferred for the proposed ML model, which is consistent with the idea where more data produces better performing models.

The maximum energy savings (26%) among all the inputs are obtained by all-Runtime for a GAMESS MSN input (msn-11) because all-Runtime uniformly executes the MSN inputs at the reduced uncore frequency without tinkering with the core frequency. The other two variants do reduce the core frequency for the MSN inputs, consequently resulting in their much lower energy savings. Overall, across all the eight inputs, the average energy savings by all-Runtime, trn-Static, and eq-PerfMod were 17.7%, 7.2%, and 9.9%, respectively.

**Figure 9** traces the change in the core frequency for the three tested variants

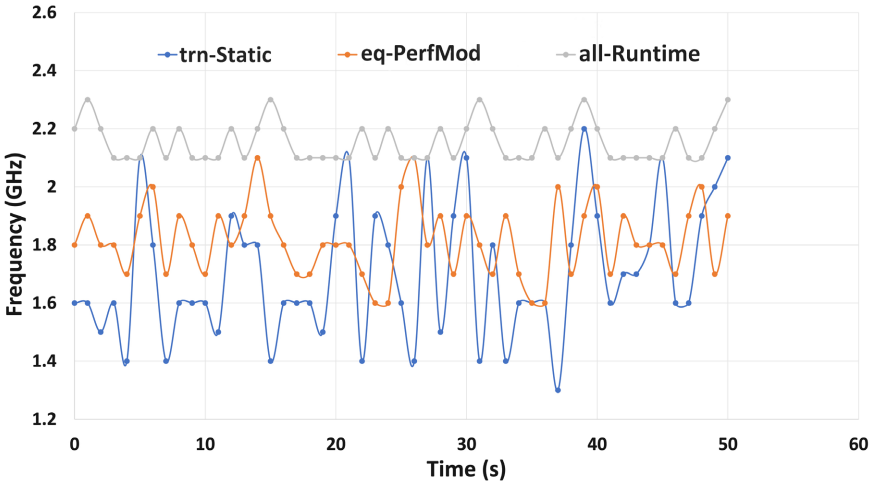


**Figure 9.** Change in core frequency for the msn-11 input during the first 50 seconds of execution.

on the msn-11 input during the first 50 seconds of its execution. It can be observed that all-Runtime keeps the core frequency at ~2.1 GHz while trn-Static and eq-PerfMod prescribe around 1.6 GHz and 1.8 GHz, respectively, thereby severely degrading the application performance. The reason for their selecting a relatively low core frequency values is that they both rely on a static performance model (static heuristic components for eq-PerfMod), which is not being dynamically updated at each timeslice as this is done in all-Runtime .

## 7. Conclusions and Future Work

In this paper, a runtime strategy using both DVFS and uncore frequency scaling is proposed to maximize energy savings for a parallel application under a given performance constraint. Machine learning based performance modeling was developed such that it incurs low overhead during its runtime usage while delivering a good prediction accuracy. Strategy variants with a training stage performed statically or dynamically were analyzed and compared with authors' previously developed strategy.

Experiments on a 28-core Haswell-EP platform with the NAS-NPB benchmarks and GAMESS MSN inputs showed that the proposed strategy provided significant energy savings with minimal performance degradation. Specifically, for an MSN input, 26% energy savings was achieved with a small 5% performance loss. Overall, a clear win of the proposed fully runtime ML model was demonstrated by its highest average energy savings of 17.7% with the lowest average performance loss of 5.1%.

Future work will focus on developing runtime power-limiting strategies driven by machine learning modeling that will maximize performance under a given power budget. The proposed runtime strategy will be further extended to multinode multi-GPU scenario where performance modeling for GPUs will be explored. Ensemble modeling options would be explored to boost the prediction

accuracy of the utilized ML models.

## Acknowledgements

This work was supported in part by the U.S. Department of Energy (DOE) Office of Science, Office of Basic Energy Sciences, Computational Chemical Sciences (CCS) Research Program under work proposal number AL-18-380-057 and the Exascale Computing Project (ECP) through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC00-07CH11358, and by the National Science Foundation under grant CNS-1828593.

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] Pagani, S., Sai Manoj, P.D., Jantsch, A. and Henkel, J. (2020) Machine Learning for Power, Energy, and Thermal Management on Multicore Processors: A Survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **39**, 101-116. https://doi.org/10.1109/TCAD.2018.2878168

[2] Barca, G.M.J., Bertoni, C., Carrington, L., Datta, Di., De Silva, N., Emiliano Deustua, J., et al. (2020) Recent Developments in the General Atomic and Molecular Electronic Structure System. *Journal of Chemical Physics*, **152**, Article ID: 154102. https://doi.org/10.1063/5.0005188

[3] Sundriyal, V., Sosonkina, M., Westheimmer, B. and Gordon, M.S. (2018) Core and Uncore Joint Frequency Scaling Strategy. *Journal of Computer and Communications*, **6**, 184-201. https://doi.org/10.4236/jcc.2018.612018

[4] Chung, E.Y., Benini, L. and De Micheli. G. (1999) Dynamic Power Management Using Adaptive Learning Tree. *IEEE/ACM International Conference on Computer-Aided Design*, *Digest of Technical Computer-Aided Design* (ICCAD-99), San Jose, 7-11 November 1999, 274-279.

[5] Tan, Y., Liu, W. and Qiu, Q. (2009) Adaptive Power Management Using Reinforcement Learning. *Proceedings of the* 2009 *IEEE/ACM International Conference on Computer-Aided Design—Digest of Technical Papers*, ICCAD 2009, San Jose, 2-5 November 2009, 461-467. https://doi.org/10.1145/1687399.1687486

[6] Wang, Y., Triki, M., Lin, X., Ammari, A.C. and Pedram, M. (2013) Hierarchical Dynamic Power Management Using Model-Free Reinforcement Learning. *Proceedings of the* 14*th International Symposium on Quality Electronic Design* (ISQED), Santa Clara, 4-6 March 2013, 170-177. https://doi.org/10.1109/ISQED.2013.6523606

[7] Jung, H. and Pedram, M. (2010) Supervised Learning Based Power Management for Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **29**, 1395-1408. https://doi.org/10.1109/TCAD.2010.2059270

[8] Cacciari, M., Bartolini, A., Tilli, A. and Benini, L. (2013) Thermal and Energy Management of High-Performance Multicores: Distributed and Self-Calibrating Model-Predictive Controller. *IEEE Transactions on Parallel and Distributed Systems*, **24**, 170-183. https://doi.org/10.1109/TPDS.2012.117

[9] Ye, R. and Xu, Q. (2014) Learning-Based Power Management for Multicore Processors via Idle Period Manipulation. *IEEE Transactions on Computer-Aided De-*

*sign of Integrated Circuits and Systems*, **33**, 1043-1055.
https://doi.org/10.1109/TCAD.2014.2305838

[10] Muhammad Mahbub Ul Islam, F. and Lin, M. (2015) A Framework for Learning Based DVFS Technique Selection and Frequency Scaling for Multi-Core Real-Time Systems. 2015 *IEEE* 17*th International Conference on High Performance Computing and Communications*, 2015 *IEEE* 7*th International Symposium on Cyberspace Safety and Security*, *and* 2015 *IEEE* 12*th International Conference on Embedded Software and Systems*, New York, 24-26 August 2015, 721-726,.

[11] Coskun, A.K., Simunic Rosing, T. and Gross, K.C. (2008) Temperature Management in Multiprocessor Socs Using Online Learning. 2008 45*th ACM/IEEE Design Automation Conference*, Anaheim, 8-13 June 2008, 890-893.
https://doi.org/10.1145/1391469.1391693

[12] Sundriyal, V. and Sosonkina, M. (2016) Joint Frequency Scaling of Processor and DRAM. *The Journal of Supercomputing*, **72**, 1549-1569.
https://doi.org/10.1007/s11227-016-1680-4

[13] Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V. and Weeratunga, S.K. (1991) The NAS Parallel Benchmarks-Summary and Preliminary Results. *Proceedings of the* 1991 *ACM/IEEE conference on Supercomputing*, Albuquerque, 18-22 November 1991, 158-165.
https://doi.org/10.1145/125826.125925

[14] Geron, A. (2019) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. 2nd Edition, O'Reilly Media, Inc., Sebastopol.

[15] Singh, R. and Singh Mangat, N. (1996) Stratied Sampling. In: *Elements of Survey Sampling*, Springer, Dordrecht, 102-144.
https://doi.org/10.1007/978-94-017-1404-4_5

[16] Zhang, Z. and Morris Chang, J. (2014) A Cool Scheduler for Multi-Core Systems Exploiting Program Phases. *IEEE Transactions on Computers*, **63**, 1061-1073.
https://doi.org/10.1109/TC.2012.283

[17] Intel (2022) Intel® 64 and IA-32 Architectures Software Developer Manuals.
https://software.intel.com/en-us/articles/intel-sdm

[18] Ioannou, N., Kauschke, M., Gries, M. and Cintra, M. (2011) Phase-Based Application-Driven Hierarchical Power Management on the Single-Chip Cloud Computer. 2011 *International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), Galveston, 10-14 October 2011, 131-142.
https://doi.org/10.1109/PACT.2011.19

[19] https://www.corsair.com/us/es/blog/DDR3-vs-DDR4