An Interpretable Machine Learning Model Enhanced Integrated CPU-GPU DVFS Governor

Article *in* ACM Transactions on Embedded Computing Systems · October 2021

DOI: 10.1145/3470974

Jurn-Gyu Park
Nazarbayev University
21 PUBLICATIONS **163** CITATIONS
SEE PROFILE

Nikil Dutt
University of California, Irvine
657 PUBLICATIONS **14,952** CITATIONS
SEE PROFILE

Sung-Soo Lim
Kookmin University
54 PUBLICATIONS **1,075** CITATIONS
SEE PROFILE

An Interpretable Machine Learning Model Enhanced Integrated CPU-GPU DVFS Governor

JURN-GYU PARK, Nazarbayev University, Kazakhstan
NIKIL DUTT, University of California, Irvine, USA
SUNG-SOO LIM, Kookmin University, South Korea

Modern heterogeneous CPU-GPU-based mobile architectures, which execute intensive mobile gaming/graphics applications, use software governors to achieve high performance with energy-efficiency. However, existing governors typically utilize simple statistical or heuristic models, assuming linear relationships using a small unbalanced dataset of mobile games; and the limitations result in high prediction errors for dynamic and diverse gaming workloads on heterogeneous platforms. To overcome these limitations, we propose an interpretable machine learning (ML) model enhanced integrated CPU-GPU governor: (1) It builds tree-based piecewise linear models (i.e., model trees) offline considering both high accuracy (low error) and interpretable ML models based on mathematical formulas using a simulatability operation counts quantitative metric. And then (2) it deploys the selected models for online estimation into an integrated CPU-GPU Dynamic Voltage Frequency Scaling governor. Our experiments on a test set of 20 mobile games exhibiting diverse characteristics show that our governor achieved significant energy efficiency gains of over 10% (up to 38%) improvements on average in energy-per-frame with a surprising-but-modest 3% improvement in Frames-per-Second performance, compared to a typical state-of-the-art governor that employs simple linear regression models.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *System on a chip*; • **Software and its engineering** → **Power management**; • **Computing methodologies** → *Graphics processors;*

Additional Key Words and Phrases: Machine learning techniques, power management policies, dynamic voltage and frequency scaling (DVFS), integrated GPU, model-based design, interpretable machine learning models

ACM Reference format:
Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. 2021. An Interpretable Machine Learning Model Enhanced Integrated CPU-GPU DVFS Governor. *ACM Trans. Embedd. Comput. Syst.* 20, 6, Article 108 (October 2021), 28 pages.
https://doi.org/10.1145/3470974

This article is an extended version of the following paper presented at the IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia), Seoul, Republic of Korea, Oct. 15–20, 2017 [25].

This work is partly supported by the Nazarbayev University (NU), under grant 021220FD0851 (FDCRGP).

Authors' addresses: J.-G. Park, Nazarbayev University, School of Engineering and Digital Sciences, Computer Science, Nur-Sultan, Kazakhstan; email: jurn.park@nu.edu.kz; N. Dutt, University of California, Irvine, School of Information and Computer Science, Irvine, CA, 92697, USA; email: dutt@ics.uci.edu; S.-S. Lim, Kookmin University, Seoul, South Korea; email: sslim@kookmin.ac.kr.

1539-9087/2021/10-ART108 $15.00
https://doi.org/10.1145/3470974

1 INTRODUCTION

Mobile games are an increasingly important application workload for mobile devices in terms of increasing number of game applications and dynamism of gaming workloads. The recent trend toward **Heterogeneous MultiProcessor Systems-on-Chip (HMPSoC)** architectures (e.g., ARM big.LITTLE with integrated GPU) attempt to meet the performance needs of mobile devices, and rely on software governors for dynamic power management in the face of high performance.Besides separate governors for contemporary commercial CPU and GPU DVFS power management, some recent research efforts have proposed integrated CPU-GPU DVFS policies [18, 29] for a small set of mobile games, assuming fairly static gaming workloads.

However, gaming applications [17] exhibit inherent dynamism in their workloads, and recent research on software governors typically use classical statistical methods (e.g., simple linear regression models [29] with a small amount of specific training data), resulting in high prediction errors for unseen workloads. These classical linear regression–based approaches are not effective for capturing the non-linear dynamism of gaming applications, since they impose a linear relationship on the data [36]. To overcome the limitations of classical statistical models (e.g., assumption of linear relationship or considering a large number of variables), some recent approaches for **General-purpose GPUs (GPGPUs)** [39] and **High-Performance Computing (HPC)** [9] have developed performance prediction models using **machine learning (ML)** techniques. To the best of our knowledge, an interpretable machine learning model enhanced approach targeting embedded systems has not been investigated for CPU-GPU integrated governors managing gaming workloads on mobile heterogeneous platforms.

To address these issues, we propose a machine learning enhanced integrated CPU-GPU DVFS governor (Figure 1) that proceeds in two phases: (1) in the learning phase, we build tree-based models with highly accurate and interpretable models using practical offline machine learning techniques, and (2) we deploy these models into an integrated DVFS governor that achieves online runtime estimation using the models. And, the objective of our governor is to achieve a target **Frames-per-Second (FPS)** or/and utilization as much as possible with minimal total power consumption using the integrated CPU-GPU DVFS.

Our article makes the following specific contributions:

- We propose an interpretable machine learning enhanced model building methodology using the scikit-learn estimators in Python [31] for highly accurate and interpretable models, using diverse and dynamic gaming datasets collected from a real heterogeneous mobile platform.
- To evaluate quantitatively the model interpretability, we propose a simple but effective metric, the simulatability (i.e., the ability of a person to simulate a built model for a given input and get the correct output [20, 35]) operation counts (**simulatability operation counts (SOC)**).
- We present an integrated CPU-GPU DVFS governor that applies piecewise policies with analyses of the models.
- We present experimental results on a set of 20 mobile games with various characteristics, showing significant energy savings of over 10% (up to 38%) in **Energy-per-Frame (EpF)** with a surprisingly concomitant 3% improvement in FPS performance, compared to a state-of-the-art governor.

The rest of the article is organized as follows: Section 2 gives motivation and related work. Section 3 distinguishes our methodology using the learning phase with the SOC formulas and prediction phases. Section 4 shows and analyzes our results. Finally, Section 5 concludes with a summary and future work.

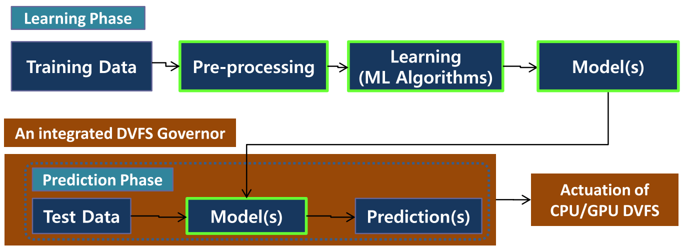


Fig. 1. Machine learning approach for our system [25].

2 MOTIVATION AND RELATED WORK

Unlike general machine learning–based approaches, embedded systems pose two crucial challenges for machine learning: (1) model building considering system specific characteristics or constraints such as heterogeneous architectures and system software layers (i.e., not just pursing higher accuracy) and (2) an interpretable structure of built models for model integration and evaluation of the integrated system, within a resource-constrained platform.

With regard to the first challenge, a specific characteristic of our integrated governor is that our prediction models should be integrated into a CPU-GPU governor, and the models must estimate the appropriate CPU and GPU frequencies for each state while achieving the system goal of maximizing energy savings with minimal performance degradation. The second challenge requires that the built models should be easily integrated into the governor algorithm, with negligible computational overhead within the time interval epochs of the CPU or GPU governors.

While there are many powerful machine learning techniques such as an instanced-based learning (e.g., **k-Nearest Neighbors (KNN)**) or neural networks that provide high accuracy, they also suffer from high opacity (not revealing anything about the structure of built models). However, the class of regression models provides high interpretability but relatively low accuracy.

2.1 Motivation

To solve the challenge of predicting real values for non-linear type datasets, we select a tree-based piecewise linear model (i.e., model trees [32, 36]), which combines a conventional decision tree with the possibility of linear regression models at the leaves. **The model trees (MT)** model is considered as a simple, accurate and robust model [3, 32, 36]; and the robustness [14] can be improved by simplifying the tree (i.e., pruning by merging some of the lower sub-trees into one node). And the experimental results [36], revealing the robustness when dealing with missing data, show that MT performs significantly better than CART [4]. The model trees provides high accuracy with an interpretable structure of built models that allows for ease of integration into the governors. We note that the integrated governor using these prediction models may sometimes lead to unexpected results such as sudden performance drops (e.g., due to over-/under-fitting of the models, unseen dynamic workloads, or heuristic thresholds in a governor algorithm).

For a practical example, consider a system software module for dynamic power management that is integrated into the operating system (Linux device driver) layer, where simple and relatively accurate built models are required for the constrained resources and reducing overheads. Moreover, a major challenge arises when the actual results on the target platforms are significantly different from our ML model predictions. What kind of attempt is needed to solve the problem? As long as the prediction is based on a model, we may try to understand this problem through the simulations of output values by using the same or changing input values and comparison with the actual results; at least these efforts are the starting point to get hints for the problem solving; and the integrated models should be interpretable.

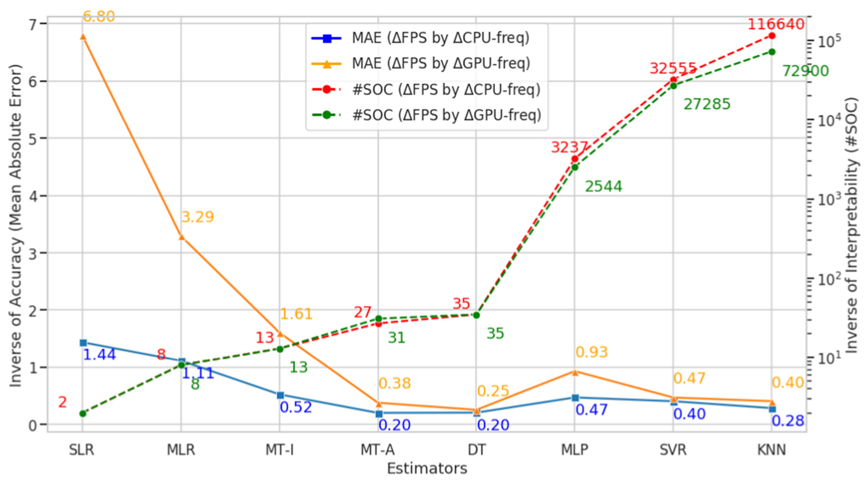


Fig. 2. Motivating example: Comparison of accuracy (inverse of MAE) and local interpretability (inverse of #SOC) among the machine learning algorithms.

For the simulations to calculate the output values, the most relevant model interpretability is the simulatability [20, 35] (i.e., the ability of a person to simulate a built model for a given input and get the correct output). To improve the simulatability, sparsity is also a useful measure of interpretability, because humans can handle at most 7 ± 2 cognitive entities at once [33]. This is one main reason that ML model selection on specific domains such as embedded systems more focuses on their interpretability (especially simulatability) even compromising their accuracy, by constraining the number of non-zero coefficients in linear models or/and the number of max depth in tree-based models. However, model selection pursing only accuracy is not able to give actionable insights due to no-sparsity or/and the black box nature of ML techniques that use deep neural networks with many hidden layers and neurons.

Therefore we found it is important to deploy a simple and interpretable model that allows us to investigate and resolve the problems in the integrated system. Note that our approach is a combination of a decision tree with regression models at leaf nodes; the representation of our models is perspicuous, because the decision tree structure is simple due to a small number (e.g., four) of leaf nodes in a tree and the regression functions do not involve many attributes due to improved feature selection.

Now consider Figure 2, where we compare model accuracy (prediction errors) and interpretability (the **number of the simulatability operation counts (#SOC)**) of the built models using the machine learning estimators/algorithms (Table 1) to motivate the need for accurate and interpretable tree-based piecewise regression models. (For the motivating example, we use a dataset, which is collected from a real platform using a training set of 45 diverse gaming applications [17], considering a test size of 0.3; and the characteristics of the dataset and platform configurations are described in detail in Sections 4.1).

The machine learning algorithms except the MTs [8, 38] described in Table 1 are already built in the scikit-learn machine learning in Python [31]. ΔFPS prediction by $\Delta CPU\text{-}Freq$ ($\Delta GPU\text{-}Freq$) in Figure 2 is FPS sensitivity to CPU (GPU) frequency change after feature selection.

Model Accuracy: As a metric for the prediction errors (the left y-axis) in terms of accuracy, we use the **Mean Absolute Error (MAE)**, which is a good indicator of average model performance [37].

Table 1. Compared M.L Estimators/Algorithms

Algorithms in scikit-learn [31]	Descriptions
SLR	Simple Linear Regression
MLR	Multivariate Linear Regression
DT	Decision Trees
MLP	Multi-layer Perceptron
SVM (SVR)	Support Vector Machines (Regression)
KNN [1, 15]	k-Nearest Neighbors
MT [8, 32, 36, 38]	Linear Model Trees

Model Interpretability: As a metric for the interpretability/complexity of built models (the right y-axis) from the perspective of interpretable ML models, we propose simple but effective formulas for calculating SOC.For details, it is described in the methodology (Section 3.1).

For the model trees, if the decision tree is simple (i.e., a small number of leaf nodes) and the regression models do not normally involve many variables after a feature selection, the representation is simple and interpretable. However, as the number of leaf nodes in a tree increases, the #SOC will also increase with less interpretability but more accurate MT model (MT-A) as shown in the middle of Figure 2. For instance, while the average MAE of **Simple Linear Regression (SLR)** and **Multivariate Linear Regression (MLR)** for Δ*FPS by* Δ*GPU-Freq* (i.e., FPS sensitivity to CPU frequency change) is 6.8 and 3.3, respectively, that of the interpretable model tree (MT-I) using 4 leaf nodes is 1.6 (i.e., the MAE of the MT-I was reduced by about 76% and 52% compared to SLR and MLR, respectively). From the perspective of model interpretability, MT-I has relatively similar-level #SOC with the multivariate linear models in Figure 2 (13 #SOC for MT-I, compared to 8 #SOC for the MLR), in addition to the accuracy improvement. This comparison clearly shows the need for an interpretable model with high accuracy; the model trees address this appropriately, with a small number of leaf nodes in a tree and highly interpretable regression functions after feature selection.

2.2 Related Work

With the emergence of high-performance integrated mobile GPUs, several research efforts have proposed integrated CPU-GPU DVFS governors: Pathania et al.'s [30] integrated CPU-GPU DVFS algorithm didn't consider quantitative evaluation for energy savings (e.g., per-frame energy or FPS per watt); their next effort [29] further developed power-performance models to predict the impact of DVFS on mobile gaming workloads, but used simple linear regression models using a small amount of specific data resulting in high prediction errors for various unseen workloads. Kadjo et al. [18] used a heuristic queuing model to capture CPU-GPU-Display interactions across a narrow range of games exhibiting limited diversity in CPU-GPU workloads.

In our previous work, we first introduced the mobile graphics workload characterization [26] analyzing the abstract mobile gaming/graphics pipeline and developing micro-benchmarks that stress specific stages of the graphics pipeline separately as workload factors, and studied the relationship between varying graphics workloads and resulting energy and performance of different mobile graphics pipeline stages. And then, we proposed a coordinated CPU-GPU maximum frequency capping technique [27] and applied it to a diverse range of mobile graphics workloads, but assumed static characterization of each application. Concurrently, we investigated the memory access footprint for graphics-intensive gaming applications, gaming/graphics threads model and cluster-based DVFS characteristics; and then we proposed a memory-aware cooperative CPU-GPU DVFS governor [16] that considers both the memory access footprint as well as the CPU/GPU frequency to improve energy efficiency targeting especially high-end (memory-intensive) mobile game workloads. Our next effort [28] further developed a **hierarchical FSM-based (HFSM)** dynamic behavior modeling strategy for mobile gaming considering QoS and CPU/GPU workload dynamism, but used an adaptive frequency-capping technique on top of the default CPU and GPU governors instead of proposing a prediction model–based integrated frequency scaling technique.

Recently, CPU/GPU performance or power estimation models that use machine learning techniques are emerging to overcome these challenges: model building from training data at numerous different hardware configurations for diverse and dynamic applications (workloads) on HPC or GPGPU platforms. Dwyer et al. [9] proposed a method for estimating performance degradation on multicore processors and applied into HPC workloads; Wu et al. [39] presented a GPGPU power and performance estimation model that uses machine learning techniques on measurements from real hardware performance counters. However, both do not consider an interpretable model structure for easy integration of the models into a system and also do not analyze the effects of the models during runtime prediction; instead these efforts focus purely on high accuracy.

Our previous preliminary effort [25] introduced a machine learning enhanced integrated CPU-GPU governor that builds tree-based piecewise linear models (i.e., model trees) considering accuracy and the structure of the built models on embedded (on-device) systems. However, this work did not propose a quantitative metric for interpretable machine learning models to quantitatively calculate SOC of the models and compare them between the built models, but used an ad hoc model structure category of simple, medium and complex.

Gupta et al. [11] [10] introduced a need for online performance models that can adapt to varying workloads, since the impact of the GPU frequency on performance varies rapidly over time and presented a light-weight adaptive runtime performance model that predicts the frame processing time; yet they did not present an integrated CPU-GPU governor but only introduced potential impacts for GPU dynamic power management using the model. Chuan et al. [6] proposed an adaptive on-line CPU-GPU governor for games on mobile devices to minimize energy consumption. However, their work was applied to a set of only three games exhibiting a narrow range of CPU-GPU workloads; and they did not show applicability across a wide range of games exhibiting diverse CPU-GPU workloads in spite of significantly different results from different types of graphics workloads.

In more recent work, energy-efficient control of mobile processors based on Long Short-Term Memory deep learning models by Lee et al. [19], online dynamic resource management of mobile platforms using **Reinforcement Learning (RL)** by Gupta et al. [12] and Imitation Learning by Mandal et al. [21] approaches were introduced. However, their work focus only on CPU dynamic power management using uninterpretable black box models. Inherently interpretable models and black box machine learning (e.g., deep learning) models [5, 23, 24] are currently being used for high-stakes decision making throughout society, to solve problems in healthcare, criminal justice and computer vision domains [33]. However, ML model selection techniques targeting embedded systems do not consider model interpretability with the lack of a quantitative metric but use only simple statistical models [29], heuristic models [18] or uninterpretable black box models [12, 19, 21], although considering interpretable ML models is crucial for embedded system software modules in terms of portability/integration and analyses for problem solving on mobile heterogeneous platforms.

To the best of our knowledge, our work—unlike previous efforts focused purely on performance (accuracy) without regard to interpretability for mobile (on-device) systems—is the first to introduce an interpretable machine-learning model-based piecewise linear model building methodology that achieves high accuracy while using analyzable model interpretability, allowing for ease of integration into CPU-GPU integrated governors for mobile platforms. We therefore aim to achieve a target FPS with minimal total power consumption using an integrated CPU-GPU DVFS. Furthermore, we present experimental results for an integrated CPU-GPU governor applied on mobile

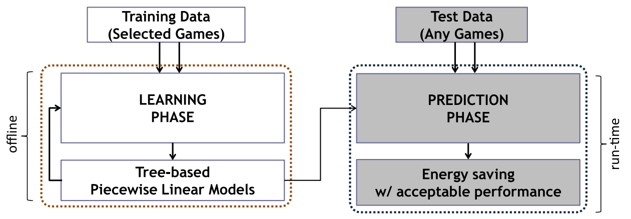


Fig. 3. Methodology overview [25].

gaming workloads using a test set of 20 mobile games exhibiting diverse characteristics executing on a mobile HMPSoC platform.

3 METHODOLOGY

Our methodology has two phases: a learning phase and a prediction phase as shown in Figure 3. In the learning phase, we build tree-based piecewise linear regression models in addition to the comparable machine learning models (Table 1), using offline machine learning techniques built in a scikit-learn machine learning in Python [31], considering both high accuracy (low error) and interpretable models using a quantitative metric, the SOC. Then in the prediction phase, the integrated governor uses the built models at runtime to estimate appropriate CPU and GPU frequencies (as much as possible maximizing energy savings with minimal FPS degradation).

The main goal of this work is to present a new practical model-building approach using offline machine learning techniques evaluating model accuracy and interpretability using the metric in the learning phase. And then we evaluate the real effects of the prediction models in terms of energy saving and performance (FPS) in the prediction phase. Therefore, for an integrated CPU-GPU governor framework, we deploy a simple but already qualified integrated governor framework using a hierarchical-FSM-based representation based on thorough observations of state-of-the-art power management techniques [28, 29].

3.1 Simulatability Operation Counts for Interpretable ML Models

As the first step of the learning phase, we present (1) the definition of interpretable ML models, (2) the overall methodology for the SOC formulas, (3) the details of the #SOC calculation in each estimator with the corresponding examples, and (4) application the SOC formulas to the motivating example (Figure 2).

Definition: We build interpretable machine learning models that are constrained in model form so that it is either useful to someone or obeys structural knowledge of the domain, such as monotonicity, structural (generative) constraints or physical constraints that come from domain knowledge [33]. And, we focus on local simulatability: the ability of a person to—independently of a computer—run a model and get the correct output for a given input [20, 35] to resolve problems from the resource constrained embedded systems, while model interpretability is a more domain specific notion, so there can not be an all-purpose definition [33] (i.e., interpretability can be differently defined according to application domains).

Methodology: To consider the #SOC of interpretable ML models, we focus on the easiness/simplicity of the two subtypes for simulatability [20]: (1) The size of the model needs to be small, and (2) the computation required to perform inference needs to be easy/simple, consuming a short time with the small number of operation counts. Our methodology follows these steps:

Table 2. Summary of Simulatability Operation Counts Formulas

Algorithm	Prediction/Inference Model for Computation Analysis	#SOC
Linear Reg.	$\hat{y_{lr}} = \sum_{p=1}^{P}(w_p \times x_p) + b$	$2 \times P$
Decision Trees	$\hat{y_{dt}} = c_m I\{(x_1, x_2, \ldots x_p) \in R_m\}$	$2 \times D + 1$
Model Trees	$\hat{y_{mt}} = \hat{y_{lr}} I\{(x_1, x_2, \ldots x_p) \in R_m\}$	$2 \times D + 2 \times P + 1$
SVR	$\hat{y_{svr}} = \sum_{i \in SV}(\alpha_i - \alpha_i^*)K(x_i, x) + b$	$(2 + K_t) \times SV$
KNN	Instance-based learning	$I \times P \times D_t$ (Distance type)
MLP (H = 1)	$\sum_{j=1}^{N_2}(w_{1j}^{(3)} \times At(\sum_{i=1}^{N_1}(w_{ji}^{(2)} \times x_i) + b_1)) + b_2$	$\sum_{h=1}^{H}(2 \times N_h + A_t) \times N_{h+1} + 2 \times N_{h+1}$
MLP (H = 2)	$\sum_{k=1}^{N_3}(w_{1k}^{(4)} \times At(\sum_{j=1}^{N_2}(w_{kj}^{(3)} \times At(\sum_{i=1}^{N_1}(w_{ji}^{(2)} \times x_i) + b_1)) + b_2)) + b_3$	
MLP (H = h)	$\hat{y_{mlp}} = \sum_{h=1}^{N_{h+1}}(w_{1h}^{(h+2)} \times At(MLP(H = h - 1)) + b_{h+1}$	

Table 3. Summary of Model Parameters and Hyper-parameters

Param.	Definition	Obtained from
P	Number of finally selected predict features	Table 6
D	Number of maximum depth in Tree models	Model Training
K_t	Type of kernel function	Table 4
SV	Number of support vectors in SVM	Model Training
I	Number of instances in Training	Dataset
D_t	Type of distance function	Table 4
H	Number of hidden layers	Hyper-parameters
N_h	Number of neurons in the h layer ($N_1 = P$)	Hyper-parameters
A_t	Type of activation function	Table 4
w_p	Weights (parameters) of the predict variables	Model Training
b	Bias	Model Training
x_p	The finally selected predict variables	Table 6
R_m	Regions of the 'm' partitions in Tree models	Model Training
c_m	Constant value in a region	Model Training

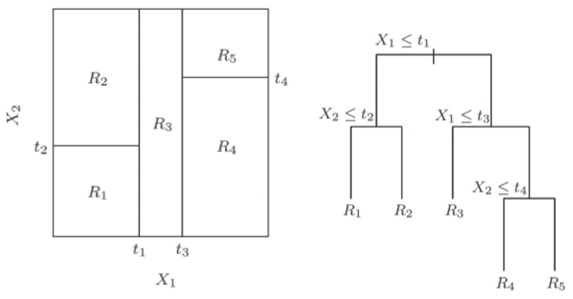
(1) Define and analyse the prediction/inference model of each estimator/algorithm (Table 2).
(2) Find the key model parameters and hyper-parameters representing the model size and computation of each estimator (Tables 3 and 4).
(3) Compute analytically the model to perform inference/prediction and calculate all the operations for the computation using from simple to complicated examples (Detailed #SOC Calculation).
(4) Inductively design generalized SOC formulas for all the estimators by analyzing the operation count patterns with the parameters and the kernel and activation types (Table 2 and 4).
(5) Verify the SOC formulas using examples of variations of the key parameters (Detailed #SOC Calculation and Table 5).

Detailed #SOC Calculation (Tables 2–4)

1. Linear Regression: If the SLR model is $\hat{y_{lr}} = w * x + b$, then the #SOC has only 2 arithmetic (one multiplication and one addition) operations, because w and b are the trained coefficients and the local input x will be given to simulate/predict the expected output $\hat{y_{lr}}$. Therefore, the SLR provides the most interpretable model with the smallest #SOC. For the MLR, the model size is dependent on the selected features P; and it is finally dependent on the number of non-zero coefficients among the selected features. If the MLR model is $\hat{y_{lr}} = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + \cdots w_p * x_p + b$, then the #SOC has $2 \times P$ (one multiplication and one addition per feature) operations, assuming all the $w_1 - w_p$ and b are already trained coefficients and all the local inputs $x_1 - x_p$ are given.

Table 4. Summary of Kernel, Distance, and Activation function Types

K_t	Kernel Type	Function	#SOC
K_{lin}	Linear	$\langle x, x' \rangle$	$2 \times P - 1$
K_{poly}	Polynomial	$(\gamma \langle x, x' \rangle + r)^d$	$K_{lin} + 2 + d(deg.)$
K_{rbf}	Radial basis function	$\exp(-\gamma \lVert x - x' \rVert^2)$	$3 \times P + \#_{exp}$ (3)
K_{sig}	Logistic sigmoid	$\tanh(\gamma \langle x, x' \rangle + r)$	$K_{lin} + 2 + \#_{exp} + 5$
D_t	Distance Type	Function	#SOC
D_{eucl}	Euclidean Distance	sqrt(sum($(x - y)^2$))	3
D_{man}	Manhattan Distance	sum(|x - y|)	3
A_t	Activation Type	Function	#SOC
A_{relu}	Rectified linear unit	f(x) = max(0, x)	1
A_{sig}	Logistic sigmoid	f(x) = $1/(1 + 1/e^x)$	$\#_{exp} + 3$
A_{tanh}	Hyperbolic tan	f(x) = tanh(x) = $\frac{(e^x - 1/e^x)}{(e^x + 1/e^x)}$	$\#_{exp} + 5$



(a) A Partition (b) The Tree

Fig. 4. A tree partition of a two-dimensional feature space by recursive binary splitting [15].

2. Decision Trees: The model $\hat{y_{dt}}$ in Table 2 can be represented to the binary tree with higher interpretability in Figure 4(b), having a specific five-region example with two max-depths. In terms of simulatability, the prediction in the decision trees is to find the region using the split-points of the input features. The tree model size is dependent on the depth (D) for the predicted region with the information of the leaf node, rather than the number of the selected features (P). Therefore, the #SOC of DT is $2 \times D + 1$ (one comparison and one check for the leaf node per depth plus one comparison in the root node) operations, having the examples of 3 #SOC in the regions 1, 2, and 3, (and 5 #SOC in the regions 4 and 5).

3. Model Trees: The model $\hat{y_{mt}}$ is a combination of a decision tree and regression models at leaf nodes. Therefore, we use the tree examples in Figure 4(b), assuming that the built regression model in the region 1 is $\hat{y_{lr}} = w * x + b$, and $\hat{y_{lr}} = w_1 * x_1 + w_2 * x_2 + b$ in region 5. Then, the #SOC for MT in region 1 has 5 (3 in the decision tree and 2 in the regression model) and 9 (5 in the decision tree model and 4 in the regression model) in region 5. If the max-depth of the decision tree is small, then the #SOC of MT is almost similar with that of the MLR (the difference is only $2D + 1$).

4. KNN: An instance-based learning, which constructs hypotheses directly from the training instances themselves and the hypothesis complexity can grow with the instances of a dataset [34]. Therefore, the #SOC of KNN is at least $I \times P \times D_t$ (#SOC for distance calculation), exampling 116,640 = 9,720 instances $\times$ 4 input features (Table 7) $\times$ 3 operations for the distance calculation) in Figure 2, assuming the Euclidean distance ($d(x, x^*) = \sqrt{\sum_{i=1}^{P} (x_i - x_i^*)^2}$) having three (one summation, one multiplication/square and one addition for the sigma) operations per feature. In short, KNN

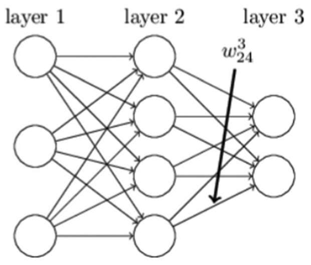


Fig. 5. A neural network notation [22].

does not reveal anything about the structure of the built model (zero-interpretability) because of the non-parametric property, except extremely small number of instances and input features.

5. SVR (SVM Regression): As shown in Table 2, the model size of the SVR is dependent on the number of support vectors ($\sum_{i \in SV}$), the coefficients of the support vectors ($\alpha_i - \alpha_i^*$) and the kernel function types ($K(x_i, x)$) with the dimensionality of the selected input features (P). And, the information of the support vectors (*support_vectors_*) and the coefficients (*dual_coef_*) can be obtained from sklearn-learn-based model selection [31]. Therefore, the #SOC can be derived to $(2 + K_t) \times SV$ from the model, depending on the SV and the kernel type (K_t) plus the two (one multiplication and one addition) operations per SV.

For the kernel type (K_t), the linear kernel computes the dot product $\langle x, x' \rangle$, depending on the number of input features (P). For example, if there are two input features (X1 and X2), then it needs 3 (two multiplication and one addition) operations ($[X1_i, X2_i][X1, X2]^T = X1_i \times X1 + X2_i \times X2$), 5 operations for 3 input features and $2 \times P - 1$ operations for P input features, shown in Table 4. For the ploynomial kernel, in addition to the K_{lin} $(2 \times P - 1)$, two operations for multiplying the γ with adding the r and d operations for the degree (power/multiplication) computation are additionally needed; for the sigmoid kernel, the two operations for r with d and the five operations for the $tanh$ function $(e^x - 1/e^x)/(e^x + 1/e^x)$ plus $\#_{exp}$. (For the $\#_{exp}$ simulation, we count the three logical operations (entering the input value, checking the output value and adopting one approximated output), assuming a simple calculator/utility can be used for this. Last for the **radial basis function (rbf)** kernel, three (one for γ and two for $||x - x'||^2$) operations per each feature were used based on the kernel function.

6. MLP: To calculate the #SOC in MLP inference model, we begin with a neural network notation [22]. We will use ($w_{ji}^{(l)}$) to denote the weight for the connection from the ith neuron in the $(l-1)$th layer to the jth neuron in the lth layer. Note that, unlike the MLP classification models, the MLP regression models have only one neuron output layer without activation function. For example, we start from one hidden layer. As shown in Table 2, the MLP prediction model has $\hat{y} = \sum_{j=1}^{N_2}(w_{1j}^{(3)} \times At(\sum_{i=1}^{N_1}(w_{ji}^{(2)} \times x_i) + b_1)) + b_2$ and #SOC (H=1) is $\sum_{h=1}^{1}(2 \times N_h + A_t) \times N_{h+1} + 2 \times N_{h+1} = (2 \times N_1 + A_t) \times N_2 + 2 \times N_2$. If we use the Figure 5 example, assuming the output layer (layer 3) has only one neuron for regression (without activation) and A_{relu} activation (1 #SOC) in the other layers, then the total #SOC is $36 = (2 \times 3 + 1) \times 4 + 2 \times 4$ ($N_1 = 3$, $N_2 = 4$). In addition, the #SOC ($H = 2$) = $\sum_{h=1}^{2}(2 \times N_h + A_t) \times N_{h+1} + 2 \times N_{h+1} = (2 \times N_1 + A_t) \times N_2 + (2 \times N_2 + A_t) \times N_3 + 2 \times N_3$ is $50 = (2 \times 3 + 1) \times 4 + (2 \times 4 + 1) \times 2 + 2 \times 2$, assuming the output layer (layer 4) has only one neuron (without activation). Likewise, any #SOC of MLP ($H = h$) can be calculated using the formula, $\sum_{h=1}^{H}(2 \times N_h + A_t) \times N_{h+1} + 2 \times N_{h+1}$ like the examples.

Table 5. Summary of Applying the SOC Formulas to the Motivating Example (Figure 2): ΔFPS Prediction by $\Delta CPU\text{-}Freq$ (ΔFPS by $\Delta GPU\text{-}Freq$)

Estimator	Formula	P (Table 6)	D	K_t or D_t	SV	I	#SOC
SLR	$2\ (2 \times P)$	1	–	–	–	–	2
MLR	$2 \times P$	4	–	–	–	–	8
MT(I)	$2 \times D + 2 \times P + 1$	4	2	–	–	–	13
MT(A)	$2 \times D + 2 \times P + 1$	4	9 (11)	–	–	–	27 (31)
DT	$2 \times D + 1$	4	17	–	–	–	35
SVR	$(2 + K_t) \times SV$	4	–	15 (K_{rbf})	1915 (1605)	–	32555 (27285)
KNN	$I \times P \times D_t$	4	–	3 (D_{eucl})	–	9720 (6075)	116640 (72900)

Estimator	Formula	N_1 = P	H	A_t	N_2	N_3	#SOC
MLP (H=2)	$\sum_{h=1}^{2}(2 \times N_h + A_t) \times N_{h+1} + 2 \times N_{h+1}$ $= (2 \times 4 + 1) \times 97 + (2 \times 97 + 1) \times 12 + 2 \times 12 = 3237$	4	2	1 (A_{relu})	97 (81)	12 (11)	3237 (2544)

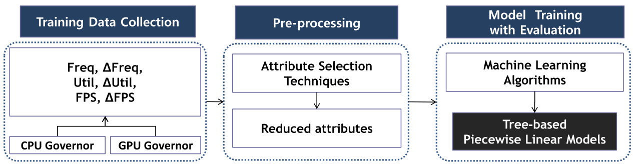

Training Data Collection
Freq, ΔFreq, Util, ΔUtil, FPS, ΔFPS
CPU Governor
GPU Governor
Pre-processing
Attribute Selection Techniques
Reduced attributes
Model Training with Evaluation
Machine Learning Algorithms
Tree-based Piecewise Linear Models

Fig. 6. Learning phase [25].

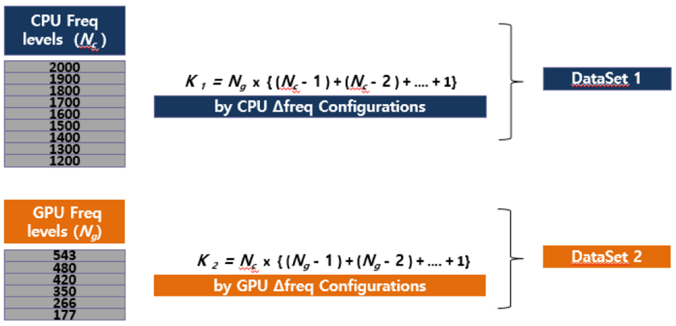
Last, for the different types of activation functions shown in the Table 4, A_{sig} has 6 (two division and one addition plus the $\#exp$) and A_{tanh} has 8 (five more for the $tanh$ function plus the $\#exp$) operations, while A_{relu} has only one operation checking whether the value is bigger than zero (>0) or not.

Applying the SOC Formulas to the Motivating Example (Figure 2): As summarized in Table 5, if we apply each formula to the Figure 2 for ΔFPS prediction by $\Delta CPU\text{-}Freq$ ($\Delta GPU\text{-}Freq$), then the #SOC for SLR is 2 and the #SOC for MLR is 8 (2×4), for the selected features (P) is 4 (Table 6). The #SOC for MT-I is 13 ($2 \times 2 + 2 \times 4 + 1$), having the two max-depth constraints in the model selection, while the #SOC for MT-A is 27 ($2 \times 9 + 2 \times 4 + 1$), having the highest accuracy in the model selection with the nine max-depth. The #SOC for DT is 35 ($2 \times 17 + 1$), having the max-depth of 17 in the model selection with the highest accuracy. The #SOC for SVR is 32,555 (27,285) = 1,915 (1,605) $\times$ 17, having the highest accuracy in the number of support vectors of 1,915 (1,605) with the rbf kernel. The #SOC for KNN is 116,640 (72,900) = 9,720 (6,075) $\times$ 3 $\times$ 4, for the number of instances of the training datasets by $\Delta CPU\text{-}Freq$ ($\Delta GPU\text{-}Freq$) is 9,720 (6,075), described in Section III.B.1, and the Euclidean distance is chosen in the model selection. The #SOC (H=2) for MLP is 3,237 = $(2 \times 4 + 1) \times 97 + (2 \times 97 + 1) \times 12 + 2 \times 12$, having the highest accuracy under two hidden layer constraints with the N_2 of 97 and N_3 of 12 (2,544 = $(2 \times 4 + 1) \times 81 + (2 \times 81 + 1) \times 11 + 2 \times 11$, having the N_2 of 81 and N_3 of 11).

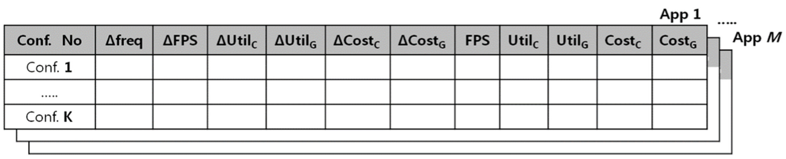
By clarifying the quantitative SOC metric for interpretable ML models in terms of the formula, calculation in each estimator and applying to the motivating example in Section 3.1, the model evaluation and selection can be processed systematically in the learning phase.

3.2 Learning Phase

As shown in Figure 6, the learning phase is composed of three steps: (1) collection of training data, (2) attribute selection, and (3) model training. The main objective of this phase is to build prediction models with simple model interpretability and high accuracy to integrate the models easily into an integrated CPU-GPU DVFS governor.


CPU Freq levels (N_c)
2000
1900
1800
1700
1600
1500
1400
1300
1200
$K_1 = N_g \times \{(N_c - 1) + (N_c - 2) + ... + 1\}$
by CPU Δfreq Configurations
DataSet 1
GPU Freq levels (N_g)
543
480
420
350
266
177
$K_2 = N_c \times \{(N_g - 1) + (N_g - 2) + ... + 1\}$
by GPU Δfreq Configurations
DataSet 2

(a) Frequency Configurations



Conf. No	Δfreq	ΔFPS	$\Delta Util_C$	$\Delta Util_G$	$\Delta Cost_C$	$\Delta Cost_G$	FPS	$Util_C$	$Util_G$	$Cost_C$	$Cost_G$
Conf. **1**											
.....											
Conf. **K**											

(b) DataSet

Fig. 7. Data collection methodology [25].

3.2.1 Collection of Training Data. For training data, we use a training set of 45 mobile applications (games and benchmarks) that has various characteristics in terms of CPU- and GPU-workloads using a workload metric (*Cost*) that is a product of the utilization and frequency [2, 27, 28, 30]; and we collect data at numerous different CPU and GPU frequency configurations using the diverse applications (workloads) on a mobile HMPSoC platform.

As shown in Figure 7, we need two datasets, because our system design has two actuators, CPU and GPU DVFS. Therefore, first we measure a raw dataset of FPS, CPU-GPU frequencies and utilizations across a range of frequency configurations by sweeping the CPU frequency across the set of frequencies (9 frequency levels in CPU, $N_c = 9$) supported by the target system at each GPU frequency of 6 frequency levels ($N_g = 6$). The number of instances for each application in the dataset can be defined using the equation, $K_1 = N_g \times \{(N_c - 1) + (N_c - 2) \dots 2 + 1\} = 6 \times 36$ in Figure 7(a); and we measure another raw dataset by sweeping the GPU frequency at each CPU frequency using the same methodology (i.e., $K_2 = N_c \times \{(N_g - 1) + (N_g - 2) \dots 2 + 1\} = 9 \times 15$). And then we collect two datasets using the values of two different CPU/GPU frequency configurations from the raw datasets assuming one is a current and the other is a new frequency. Specifically, we choose 11 crucial variables (the first row in Figure 7(b)) based on comprehensive observations of related work [18, 27–30] for mobile gaming characteristics to CPU/GPU frequency: (1) Relationship between FPS and CPU/GPU frequency. (2) Relationship between a component's utilization and its frequency. (3) The impact of cross-component frequency variations on utilizations. (4) The impacts of CPU-GPU *Cost* functions. (5) The impacts of current FPS, CPU-GPU utilizations and *Cost* functions.

As a result, in the training phase using a training set of 45 applications ($M = 45$ in Figure 7(b)), we collect two datasets of 9,720 ($45 \times 6 \times 36$) and 6,075 ($45 \times 9 \times 15$) instances, with 216 (6×36) for CPU and 135 (9×15) for GPU different frequency configurations.

3.2.2 Attribute Selection. Before training the models, we reduce the number of attributes in the datasets for two reasons: (1) to remove the attributes that are redundant or unrelated to an output,

Table 6. Selected Variables after Attribute Selection [25]

Response Variable	Selected Predictor Variables
ΔQ by ΔF_C	ΔF_C, Q, U_C and C_G
ΔU_C by ΔF_C	ΔF_C, Q, U_G, C_C and C_G
ΔU_G by ΔQ_{F_C}	ΔQ
ΔQ by ΔF_G	F_G (ΔF_G), Q, U_C and U_G
ΔU_G by ΔF_G	ΔF_G and C_C
ΔU_C by ΔQ_{F_G}	ΔQ and C_G

and (2) to reduce the complexity of built models and computation overhead, while maintaining a good prediction accuracy. Choosing a specific technique for attribute selection can be dependent on the data and application area. For our model (i.e., FPS and Utilization prediction by CPU and GPU frequency changes), the most important factor to consider is the correlation between one response attribute and the other attributes. We tested several attribute selection techniques by constructing models using the two datasets (Figure 7) and comparing their accuracy; the technique that achieved the lowest error rate was *correlation-based feature subset attribute selection (CfsSubset)* [13], which sorts the attributes by their correlation to a class attribute (response variable) and to the other attributes (predictor variables) in the dataset. Note that the number of selected predictor variables may be different from each response variable by the property of *CfsSubset* algorithm; Table 6 shows the results of attribute selection.

3.2.3 Model Training. Next, we build and evaluate models for the qualified algorithms introduced in Table 1. Here we detail the analyses and choices made for model evaluation and training. First, to build models for the six responsive variables in Table 6, we mainly train the SLR, MLR, and MT-I among the algorithms described in the motivating example (Figure 2). We note that instance-based learning does not reveal anything about the structure of built models due to the property of a non-parametric method; and that decision (regression) trees approximating a non-linear function by discretizing hundreds of leaves also are not adequate for our integrated governor. Furthermore, the highest accuracy model tree (MT-A) is evaluated for comparing model accuracy (but not for using the built models), since it is almost impossible to integrate dozens of model trees into the CPU-GPU governor and analyze the effects of the models. We then build models of MT-I by cross-validating *min_samples_leaf* (the minimum number of samples allowed at a leaf node), configuring N hyper-parameters (e.g., using *np.linspace(S, Y.shape[0] / K, N, dtype=int)* [31] with N configurations in [S,Y.shape[0]/K]), in the MT algorithm under the maximum depth of 2 (considering the number leaf nodes increase exponentially according the max maximum depth according to the tree property), which results in the number of 4 leaf nodes, because it has a simple (interpretable) model structure while its prediction errors are significantly lower than those of MLR.

Model Selection: Tables 7 and 8 summarize the results of the model interpretability and accuracy respectively. In addition, our model selection process is described in Algorithm 1.

First, from all the built models, we select candidate models in terms of a #SOC threshold ($\theta 1$), the value of 15 (lines 3 and 4), which is chosen considering simulatability and sparsity issues on embedded systems (humans can handle at most 7(±2) cognitive entities at once in the simulations, and generally each entity such as feature or depth includes two operations and one comparison in the tree root node); and additionally the minimum mean absolute error (min_MAE) among the candidate models is recorded. Second, we iteratively check accuracy level among the candidate models until the model accuracy difference (MAE_diff) is less than a threshold ($\theta 2$), which is chosen to 5, increasing model interpretability more within the tolerable 5% difference in MAE.Although the

Table 7. **INTERPRETABILITY**: #SOC Using the Formula (Table 2), Selected Features (Table 6), and Max_depth in MTs (2 for MT-I and D for MT-A)

Response Var.	SLR	MLR (P)	MT-I (D = 2)	MT-A (D)	**Candidate Models**
#SOC	2	2P	2P + 5	2P + 2D + 1	(#SOC <= $\theta 1$, 15)
ΔQ by ΔF_C	**2**	**8** (4)	**13**	27 (9)	SLR, MLR, and MT-I
ΔU_C by ΔF_C	**2**	**10** (5)	**15**	31 (10)	SLR, MLR, and MT-I
ΔU_G by ΔQ_{F_C}	**2**	**2** (1)	**7**	23 (10)	SLR, MLR, and MT-I
ΔQ by ΔF_G	**2**	**8** (4)	**13**	31 (11)	SLR, MLR, and MT-I
ΔU_G by ΔF_G	**2**	**4** (2)	**9**	19 (7)	SLR, MLR, and MT-I
ΔU_C by ΔQ_{F_G}	**2**	**4** (2)	**9**	25 (10)	SLR, MLR, and MT-I

Table 8. **ACCURACY**: Prediction Errors for Model Evaluation (10-fold Cross-validation Results, Normalizing the Prediction Errors (MAE) using the Maximum 60 FPS for Q (ΔQ by ΔF_C and ΔQ by ΔF_G)

Response Var.	SLR	MLR	MT-I	MT-A	**Chosen** (MAE_diff < $\theta 2$, 5)
ΔQ by ΔF_C	1.44 (2.40%)	1.11 (1.85%)	**0.52 (0.87%)**	0.20 (0.33%)	**MT-I (1)**
ΔU_C by ΔF_C	3.13	2.77	**2.35**	1.97	**MT-I (2)**
ΔU_G by ΔQ_{F_C}	**0.82**	0.82	0.81	0.82	**SLR 1**
ΔQ by ΔF_G	6.80 (11.33%)	3.29 (5.48%)	**1.61 (2.68%)**	0.38 (0.63%)	**MT-I (3)**
ΔU_G by ΔF_G	7.73	6.66	**5.19**	4.83	**MT-I (4)**
ΔU_C by ΔQ_{F_G}	**6.34**	6.35	6.17	4.74	**SLR 2**

For the other response variables, the MAE and the normalization are the same using 100 Utilization for U_C and U_G).

two threshold values are chosen by the specific requirements/constraints of our problem domain, they can be adjustable according to the requirements of application domains.

ALGORITHM 1: The Model Selection Process

```
1: Sort the built model list M by increasing order of #SOC

2: for each model i in M do
3:     if #SOC is equal to or less than θ1 then
4:         add model i to the candidate model list C
5:         find min_mae
6:     end if
7: end for

8: mae_diff = 100
9: while mae_diff >= θ2 and i in C do
10:        mae_diff = (curr_mae - min_mae)/min_mae * 100
11:        select the current model
12: end while
```

The last column of Table 8 shows our final model selection, comprising four MTs (1–4) and two SLRs (1–2) for the six responsive variables. These selections were made based on both higher interpretability (lower #SOC, within the threshold $\theta 1(15)$) and their lower prediction errors: Model prediction errors for ΔU_G by ΔQ_{F_C} and ΔU_C by ΔQ_{F_G} are almost similar for all the candidate models within the threshold $\theta 2(5\%)$ so that we use the simplest SLR models. However, for the other response variables, we use the MTs (1–4) because of high accuracy and interpretability.

Table 9. The Decision Tree of the MT 1

Model Tree MT 1: ΔQ by ΔF_C
1: U_C <= 93.0
2: | U_C <= 79.4 : LM1 (6345/9720)
3: | U_C > 79.4 : LM2 (1457/9720)
4: U_C > 93.0 :
5: | Q <= 44.5 : LM3 (1021/9720)
6: | Q > 44.5 : LM4 (897/9720)

Table 10. The Linear Models of the Model Tree 1

Attributes	ΔF_C	Q	U_C	C_G	Intercept
Parameters	p_1^{MT1}	p_2^{MT1}	p_3^{MT1}	p_4^{MT1}	p_5^{MT1}
LM1 ($p = \alpha$)	0.0001 ($\equiv$ 0)	−0.0138	0.0082	0.0012	0.2743
LM2 ($p = \beta$)	0.0018	−0.1143	0.0700	−0.0042	0.5862
LM3 ($p = \gamma$)	0.0151	0.0045	−0.6133	−0.0236	59.1521
LM4 ($p = \theta$)	0.0049	−0.2344	−0.1321	−0.0026	26.0564

Table 11. The Decision Tree of the MT 2

Model Tree 2: ΔU_C by ΔF_C
1: C_C <= 8.4
2: | Q <= 35.0 : LM1 (969/9720)
3: | Q > 35.0 : LM2 (110/9720)
4: C_C > 8.4
5: | C_C <= 56.3 : LM3 (5356/9720)
6: | C_C > 56.3 : LM4 (3285/9720)

Table 12. The Linear Models of the Model Tree 2

Attributes	ΔF_C	Q	U_G	C_C	C_G	Intercept
Parameters	p_1^{MT2}	p_2^{MT2}	p_3^{MT2}	p_4^{MT2}	p_5^{MT2}	p_6^{MT2}
LM1 (p = α)	−0.0035	−0.0832	0.4148	−0.5031	0.0284	−38.9406
LM2 (p = β)	−0.0066	1.8729	1.1290	−1.1974	−0.3058	−159.9498
LM3 (p = γ)	−0.0176	0.0659	−0.0569	−0.0871	0.0537	1.1211
LM4 (p = θ)	−0.0075	−0.1850	−0.0698	0.0523	0.0414	7.2508

Table 13. The Decision Tree of the MT 3

Model Tree 3: ΔQ by ΔF_G
1: U_G <= 98.9 :
2: | Q <= 53.2 : LM1 (1337/6075)
3: | Q > 53.2 : LM2 (2432/6075)
4: U_G > 98.9 :
5: | Q <= 39.4 : LM3 (1619/6075)
6: | Q > 39.4 : LM4 (687/6075)

Table 14. The Linear Models of the Model Tree 3

Attributes	ΔF_G	Q	U_C	U_G	Intercept
Parameters	p_1^{MT3}	p_2^{MT3}	p_3^{MT3}	p_4^{MT3}	p_5^{MT3}
LM1 ($p = \alpha$)	0.0116	−0.1201	−0.1653	0.0083	19.9225
LM2 ($p = \beta$)	0.0007 ($\equiv$ 0)	−0.7208	−0.0058	0.0024	43.3850
LM3 ($p = \gamma$)	0.0802	0.1188	−0.2074	0.1784	−16.2120
LM4 ($p = \theta$)	0.0241	−0.6081	−0.0696	1.4314	−104.7347

We illustrate the structure of the Model Trees using MT 1–4. For the regression coefficients, we use α, β, γ, and θ, respectively (four leaf nodes for the *max_depth* of 2), for each regression function (i.e., LM#) with the numbering for the different coefficients. Moreover, the trained parameters are determined using the MT estimator [38], which is a combination of the CART [4] plus a graph visualization tool (to quickly visualize how a model tree could prove more useful than regular CARTs) and the linear regression (i.e., *sklearn.linear_model. LinearRegression* [31]) algorithms. For example, the DT (Table 9) thresholds of the parameters (U_C and Q) and the #samples in each LM are obtained through the graph visualization tool using the *graphviz* [31] python module; and the parameters (p_{1-5}^{MT1}) of the linear models (Table 10) can be read from the *coef_* and *intercept_* attributes of the *LinearRegression* estimator.

The Model Tree MT 1 is a tree-based piecewise linear regression model to predict ΔQ by ΔF_C. Lines 1–6 reveal a tree structure with the thresholds of U_C and Q (Table 9), and Table 10 corresponds to each regression model in each leaf node with the parameters.

The Model Tree 2 is a model to predict ΔU_C by ΔF_C, revealing the four leaf nodes by the thresholds of C_C and Q (Table 11 and 12).

The Model Tree 3 is a model to predict ΔQ by ΔF_G, revealing the four leaf nodes by the thresholds of U_G and Q (Tabled 13 and 14).

Table 15. The Decision Tree of the MT 4

Model Tree 4: ΔU_G **by** ΔF_G
1: C_C <= 14.9 :
2: | ΔF_G <= 227.0 : LM1 (1075/6075)
3: | ΔF_G > 227.0 : LM2 (306/6075)
4: C_C > 14.9 :
5: | ΔF_G <= 89.0 : LM3 (1682/6075)
6: | ΔF_G > 89.0 : LM4 (3012/6075)

Table 16. The Linear Models of the Model Tree 4

Attributes	ΔF_G	C_C	Intercept
Parameters	p_1^{MT4}	p_2^{MT4}	p_3^{MT4}
LM1 ($p = \alpha$)	−0.0293	−0.2214	3.3292
LM2 ($p = \beta$)	−0.0744	−1.1698	15.9104
LM3 ($p = \gamma$)	−0.3107	−0.0539	17.9842
LM4 ($p = \theta$)	−0.1228	−0.0078	4.6170

Table 17. The SLR 1

SLR 1: ΔU_G **by** ΔQ_{F_C}
ΔU_G = 0.8786 * ΔQ + 0.0383 (we set 0.8786 = α_1^{SLR1}, 0.0383 = α_2^{SLR1})

Table 18. The SLR 2

SLR 2: ΔU_C **by** ΔQ_{F_G}
ΔU_C = 1.7289 * ΔQ - 0.3761 (we set 1.7289 = α_1^{SLR2}, -0.3761 = α_2^{SLR2})

The Model Tree 4 is a model to predict ΔU_G by ΔF_G, revealing the four leaf nodes by the thresholds of C_C and ΔF_G in Table 15 and 16 (if ΔF_G is included in the thresholds like the Lines 2 and 3 and Lines 5 and 6, we estimate it using another multivariate linear regression model). And, the SLRs for the two responsive variables (ΔU_G by ΔQ_{F_C} and ΔU_C by ΔQ_{F_G}) are as follows (Tables 17 and 18).

Model Equations: We derive model equations from the four model trees and the two SLRs. We denote the current CPU-GPU frequency combination with (F_C, F_G), the utilization values with $U_C^{(F_C,F_G)}$, $U_G^{(F_C,F_G)}$, the FPS at the frequency combination with $Q^{(F_C,F_G)}$ and the current CPU-GPU *Cost* with C_C, C_G. For each model tree, which has four leaf nodes with the *max_depth* of 2, p is α, β, γ and θ respectively for LM 1, 2, 3, and 4.

To estimate the FPS at a CPU (GPU) frequency level F'_C (F'_G), the relationship between FPS and CPU (GPU) frequency can be derived by using Model Tree 1 (Model Tree 3) as follows (ΔF_C equals F'_C - F_C, and ΔF_G equals F'_G - F_G).

$$Q^{(F'_C,F_G)} - Q^{(F_C,F_G)} = \sum_{p=1}^{4} \left(w_p^{MT1} \times x_p^{MT1}\right) + b^{MT1}, \tag{1}$$

$$Q^{(F_C,F'_G)} - Q^{(F_C,F_G)} = \sum_{p=1}^{4} \left(w_p^{MT3} \times x_p^{MT3}\right) + b^{MT3}, \tag{2}$$

To estimate the utilization of a component at a CPU (GPU) frequency level F'_C (F'_G), the relationship between a component's utilization and its frequency can be derived as follows, by using the Model Tree 2 (Model Tree 4).

$$U_C^{(F'_C,F_G)} - U_C^{(F_C,F_G)} = \sum_{p=1}^{5} \left(w_p^{MT2} \times x_p^{MT2}\right) + b^{MT2}, \tag{3}$$

$$U_G^{(F_C,F'_G)} - U_G^{(F_C,F_G)} = \sum_{p=1}^{2} \left(w_p^{MT4} \times x_p^{MT4}\right) + b^{MT4}, \tag{4}$$

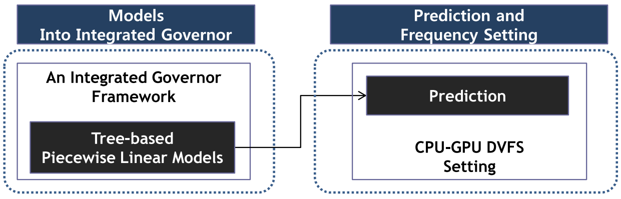


Fig. 8. Prediction phase [25].

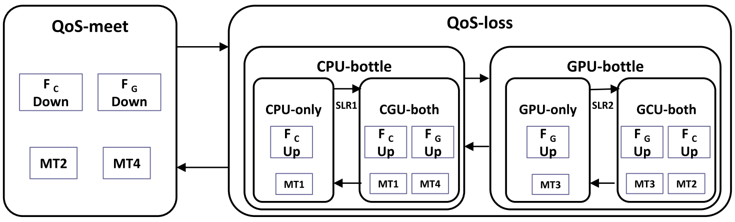


Fig. 9. HFSM-based power management Algorithm [25].

Finally, we estimate the impact of cross-component frequency variations on utilizations, which occurs as long as there is parallel increase in FPS [29]. Therefore, the corresponding equations can be derived as follows, by using the SLR2 (ΔU_C by ΔQ_{F_G}) and SLR1 (ΔU_G by ΔQ_{F_C}).

$$U_C^{(F_C,F'_G)} - U_C^{(F_C,F_G)} = \alpha_1^{SLR2}(Q^{(F_C,F'_G)} - Q^{(F_C,F_G)}), \tag{5}$$

$$U_G^{(F'_C,F_G)} - U_G^{(F_C,F_G)} = \alpha_1^{SLR1}(Q^{(F'_C,F_G)} - Q^{(F_C,F_G)}). \tag{6}$$

3.3 Prediction Phase

As shown in Figure 8, the prediction phase comprises two steps: (1) merging the models into an integrated CPU-GPU DVFS governor framework and (2) setting CPU and GPU frequencies by executing these predictors at runtime.

3.3.1 An Integrated Governor Framework. We build our governor framework on the non-trivial observations gleaned from the related work [27–29]: It is more power/energy efficient to run at higher utilization and lower frequency than lower utilization and higher frequency if a current FPS achieves a target FPS (or a current FPS is already at a quantitatively competitive performance). However, for a target-FPS–based manager, the challenge is how to get the information of a target maximum FPS or target reference values such as maximum CPU and GPU utilizations for quantitative comparison with other governors. To solve this issue (similar to the previous work [29]), we take three samples (one second duration for each) to obtain the game specific reference constants ($\hat{Q}$, $\hat{U}c$, $\hat{U}g$) when a new scene starts with the assumption that start of a scene can be detected by changes in rendered textures [7] or CPU-GPU utilization patterns [30]. This kind of sampling method enables us to avoid prior comprehensive offline profiling [29], especially to compare with the default separate CPU and GPU governors (i.e., performance-driven policy without CPU-GPU cooperation).

Figure 9 illustrates our power management algorithm using the Hierarchical FSM-based representation [28], since it provides a natural and intuitive design abstraction.

Our algorithm detects two possible super-states at current frequency combination (F_C, F_G): if Q is within $\hat{Q}$, *QoS-meet* or if Q is less than $\hat{Q}$, *QoS-loss*. For *QoS-meet* super-state, CPU and GPU frequencies will be scaled down to the estimated frequencies using the tree-based linear models to achieve the maximum utilizations (using MT 2 and 4). For *QoS-loss* super-state, there are two sub-states: If CPU is the bottleneck, then *CPU-bottle,* or if GPU is the bottleneck, then *GPU-bottle.* *CPU-bottle* sub-state has two leaf states: *CPU-only* and *CGU-both.* For *CPU-only* leaf state, only CPU frequency will be scaled up to the estimated frequency to achieve the maximum target-FPS (using MT 1); for *CGU-both* leaf state, GPU frequency will be scaled up to the estimated frequency (using MT 4) in addition to the CPU frequency scale up. (Vice versa for *GPU-bottle* sub-state: For *GPU-only* leaf state, only GPU frequency will be scaled up to the estimated frequency to achieve the maximum target-FPS (using MT 3); for *GCU-both* leaf state, CPU frequency will be scaled up to the estimated frequency (using MT 2) in addition to the GPU frequency scale up.)

3.3.2 CPU/GPU Frequency Setting with Prediction.

Performance Demand State: For the *QoS-loss* super-state, the sub-state is *CPU-bottle* or *GPU-bottle*, which require an increase in the frequency of the bottleneck component for FPS improvement. Let the needed frequency combination be (F'_C, F'_G), where $F'_C \geq F_C$ and $F'_G \geq F_G$. For the *CPU-only* leaf-state, we choose F'_C using Equation (7), derived from Equation (1): For LM 2, the equation predicts a next CPU frequency achieving the maximum target-FPS. However, if ΔF_C is not critical variable to the FPS sensitivity, then we apply an adaptive heuristic policy (i.e., one step higher (++) frequency) for LM1 and LM3. According to our observations, when we apply the same CPU frequency instead of one step higher frequency, some intensive applications result in significant performance degradation,

$$F'_C = \begin{cases} F_C + \dfrac{(\hat{Q} - Q^{(F_C,F_G)}) - \left(\sum_{p=2}^{4}\left(w_p^{MT1} \times x_p^{MT1}\right) + b^{MT1}\right)}{w_1^{MT1}} & \text{if LM} = 2, 3, 4. \\ F_C ++ \quad \text{otherwise (i.e., } w_1^{MT1} \text{ is equivalent to zero).} \end{cases} \tag{7}$$

This increase in FPS may force the cross-component (GPU) to do more work increasing its utilization; and even after increasing CPU frequency, it may fail to achieve $\hat{Q}$ because of an intermediate GPU bottleneck. The estimated U_G at $\hat{Q}$ would be given by the Equation (8), based on Equation (5),

$$U_G^{(F'_C,F_G)} = U_G^{(F_C,F_G)} + \alpha_1^{SLR1}(\hat{Q} - Q^{(F_C,F_G)}). \tag{8}$$

If $U_G^{(F'_C,F_G)}$ is greater than $\hat{U}_G$, then GPU will also become a bottleneck and the state will change to *CGU-both* leaf-state; and the GPU frequency should be increased to F'_G given by Equation (9), derived from Equation (4) (the Model Tree 4),

$$F'_G = F_G + \frac{(U_G^{(F_C,F_G)} - \hat{U}_G) - \sum_{p=2}^{2}(w_p^{MT4} \times x_p^{MT4}) + b^{MT4})}{w_1^{MT4}}. \tag{9}$$

For *GPU-bottle* sub-state, first we estimate F'_G using Equation (2) (the Model Tree 3) for *GPU-only.* Second, we can detect the condition to *GCU-both* leaf-state using Equation (6) in SLR 2. And then, F'_C can be estimated using Equation (3) (the Model Tree 2) for *GCU-both* leaf-state.

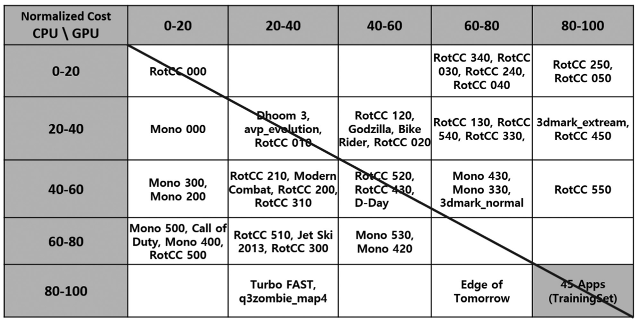
Power Saving State: For the *QoS-meet* super-state, already the maximum target-FPS is achieved with over-provisioned CPU and GPU frequencies wasting power consumption. Therefore, we can save power without quality loss by reducing CPU and GPU frequencies to F''_C and F''_G achieving the maximum CPU and GPU utilizations: Equation (10) is derived from Equation (3) (the Model Tree 2), and Equation (11) is derived from Equation (4) (the Model Tree 4),

$$F''_C = F_C + \frac{\left(\hat{U}_C - U_C^{(F_C,F_G)}\right) - \left(\sum_{p=2}^{5}\left(w_p^{MT2} \times x_p^{MT2}\right) + b^{MT2}\right)}{w_1^{MT2}}, \tag{10}$$

$$F''_G = F_G + \frac{\left(\hat{U}_G - U_G^{(F_C,F_G)}\right) - \left(\sum_{p=2}^{2}\left(w_p^{MT4} \times x_p^{MT4}\right) + b^{MT4}\right)}{w_1^{MT4}}. \tag{11}$$

Table 19. Platform Configuration [25]

Feature	Description
Device	ODROID-XU3
SoC	Samsung Exynos5422
CPU	Cortex-A15 2.0 GHz and Cortex-A7 Octa-core CPUs
GPU	Mali-T628 MP6, 543 MHz
System RAM	2-Gbyte LPDDR3 RAM at 933 MHz
Mem. Bandwidth	up to 14.9 GB/s
OS(Platform)	Android 4.4.2
Linux Kernel	3.10.9



Normalized Cost CPU \ GPU	0-20	20-40	40-60	60-80	80-100
0-20	RotCC 000			RotCC 340, RotCC 030, RotCC 240, RotCC 040	RotCC 250, RotCC 050
20-40	Mono 000	Dhoom 3, avp_evolution, RotCC 010	RotCC 120, Godzilla, Bike Rider, RotCC 020	RotCC 130, RotCC 540, RotCC 330,	3dmark_extream, RotCC 450
40-60	Mono 300, Mono 200	RotCC 210, Modern Combat, RotCC 200, RotCC 310	RotCC 520, RotCC 430, D-Day	Mono 430, Mono 330, 3dmark_normal	RotCC 550
60-80	Mono 500, Call of Duty, Mono 400, RotCC 500	RotCC 510, Jet Ski 2013, RotCC 300	Mono 530, Mono 420		
80-100		Turbo FAST, q3zombie_map4		Edge of Tomorrow	45 Apps (TrainingSet)

Fig. 10. The 45-app training set [17].

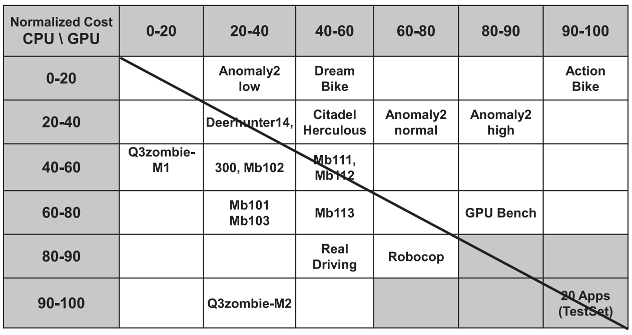
4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We evaluated our manager on the ODROID-XU3 development board installed with Android 4.4.2 and Linux 3.10.9; Table 19 summarizes our platform configuration. The platform is equipped with four TI INA231 power sensors measuring the power consumption of **big CPU cluster (CPU-bc)**, **little CPU cluster (CPU-lc)**, GPU and memory, respectively. The CPU supports cluster-based DVFS at nine frequency levels (from 1.2 GHz to 2.0 GHz) in CPU-bc and at seven frequency levels (from 1.0 GHz to 1.6 GHz) in CPU-lc, and GPU supports six frequency levels (from 177 MHz to 543 MHz).

Benchmark Set: We use a training set of 45 gaming applications and a test set of 20 unseen games including several micro-benchmarks [17, 26, 27], covering different types of graphics workloads (CPU-, GPU-, CPU&GPU-dominant and Memory-mixed) with a test size of 0.3 (i.e., 20/65 apps). Figures 10 and 11 summarize the 45 training games and the 20 test set games used in our experiments. Note that these games were selected specifically to exercise different combinations of CPU- and GPU-intensive workloads and included not only a large number of real games but also some custom micro-benchmarks to cover the entire space of different/balanced graphics workloads.

The CPU-GPU graphics workload variation and their relative intensity are typically quantified using a cost metric that is a product of the utilization and frequency [2, 27, 30]. Accordingly, we deploy normalized CPU and GPU costs, which are defined as the product of the processor current average utilization and its current average frequency divided by the product of the maximum utilization and its maximum frequency to have the normalized range of 0 to 100, as shown by the rows and columns of the tables covering most entries in Figures 10 and 11. We note that the gaming applications [17] located on the right-side of the diagonal line represent a GPU-dominant workload (since GPU cost is higher than CPU cost); similarly, games on the left of the diagonal line are CPU-dominant. Furthermore, higher values of the cost ratio represent more intensive



Normalized Cost CPU \ GPU	0-20	20-40	40-60	60-80	80-90	90-100
0-20		Anomaly2 low	Dream Bike			Action Bike
20-40		Deerhunter14,	Citadel Herculous	Anomaly2 normal	Anomaly2 high	
40-60	Q3zombie-M1	300, Mb102	Mb111, Mb112			
60-80		Mb101 Mb103	Mb113		GPU Bench	
80-90			Real Driving	Robocop		
90-100		Q3zombie-M2				20 Apps (TestSet)

Fig. 11. The 20-app test set [17, 25].

Table 20. Governors for Comparison [25]

Governor	Description
Default	Interactive CPU and proprietary GPU governor
PAT15 [29]	Simple Linear Regression-based governor
ML-Gov [25]	Our Optimized Tree-based piecewise Linear Regression (MT) governor
Pure-Adap [28]	Pure step by step higher/lower Adaptive governor
Pure-MT	Only Tree-based Linear Models w.o Adaptation governor

CPU or GPU workloads, with the highest representing the most CPU- or GPU-intensive gaming applications or benchmarks.

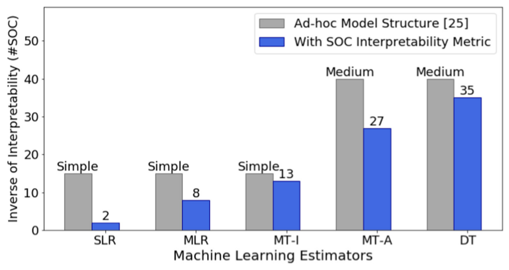
In this context, the 20 test set gaming applications analyzed in Figure 17 can be interpreted using the cost matrix model in Figure 11; we attempted to test games exhibiting a variety of CPU- and GPU- intensity, as summarized below:

- CPU-Domi + GPU-Mix applications such as mb111, mb112, and Robocop have higher GPU cost values compared to CPU-intensive workloads such as mb101, mb103, and Q3zombie-M2.
- For Mem-Mix applications, the CPU-GPU cost matrix model does not include the memory cost values. Therefore, we additionally present the memory cost values of Mem-Mix applications such as mb103 (62%), mb113 (72%), and Robocop (77%), compared to non-memory intensive applications such as mb101 (37%) and mb111 (39%).

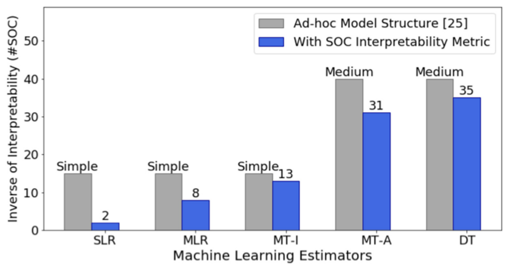
Automatic Measurement Tool: To automatically measure a large set of mobile games quantitatively, we developed an automatic measurement tool implemented using Python modules, xml files, and Linux shell scripts. Using this tool, we captured the average values of FPS, total power (CPU-bc, CPU-lc, and GPU) and EpF for three runs (120 s for each run) of each benchmark, and averaged their measurements.

For Comparison: We compare our manager against the default, a linear prediction model [29], a hierarchical pure-adaptive model [28] and a pure-MT model-based governors as shown in Table 20.

The default corresponds to the independent CPU and GPU governors in Linux (Interactive CPU governor and ARM's Mali Midgard GPU governor). The linear prediction model governor [29] (represented as PAT15) proposed an integrated CPU-GPU DVFS strategy by developing predictive simple linear regression power-performance models. The pure-adaptive governor (represented as Pure-Adap) does not use the built model trees but adaptively scale up step by step higher (or down step by step lower) frequency for corresponding component according to each leaf state of the hierarchical FSM model; and it is the key idea and methodology of our related work [28] (Hierarchical FSM-based Adaptive Integrated CPU-GPU Frequency Capping Governor). The pure model-tree governor is represented as Pure-MT, which does not use the heuristic adaptation (i.e.,



(a) Δ FPS by ΔCPU-Freq



(b) Δ FPS by ΔGPU-Freq

Fig. 12. Inter-model evaluation results on interpretability.

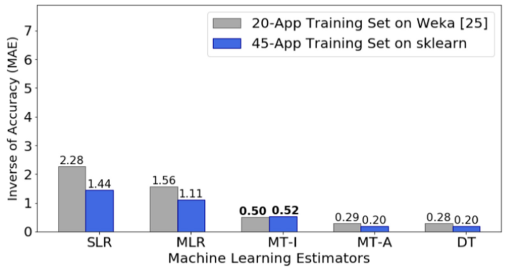
one step higher or lower) but utilizes the built model trees based on the assumption that offline learning models generalize all possible workloads that would be executed by the system. In other words, if ΔF_C or ΔF_G is not related to a response variable in the models, then we set the next frequency same with the current frequency (i.e., $F'_C = F_C$ or $F'_G = F_G$) for any states without using the heuristic adaptation.

Overhead and Epoch of Our Manager: Our power manager was implemented in the Linux kernel layer. The execution time of our manager per epoch is within 20 μs, which is totally negligible compared to the epoch period (200 ms) in terms of performance (FPS degradation). Specifically, the runtime data sampling epoch is the same with that of the default GPU governor (100 ms), and we use the average values of the default two consecutive epochs, because our governor set CPU and GPU frequencies every 200-ms epoch. And, the reason we use the epoch of 200 ms (double of the GPU governor epoch) is that a longer epoch may affect performance degradation because of delayed frequency settings while it can provide more accurate FPS information (for below 60 FPS) per epoch, and that a shorter epoch may affect more often frequency fluctuations unlike utilization-based governors or rendering glitches. Moreover, power consumption overhead can be verified by checking the difference of power consumption between with and without power manager code snippets. And, we could not observe any difference (not noticeable) between the two measurements. In addition to reason of negligible power consumption impact of 0.01% temporal overhead (instantaneous at most 20 μs per 200 ms), empirically the most power-intensive *anomaly2high* application has an average (three runs) 3329.29 mW with the standard deviation of σ = 26.94, and the confidence level (margin of error) of 95% is 3329.29 ± 30.48 mW; and the least power-intensive *mb101* application has average 1251.72 mW (σ = 12.5), and the confidence level of 95% is 1251.72 ± 14.13 mW. The not-noticeable power consumption overhead totally can be hidden in the power margins of error.

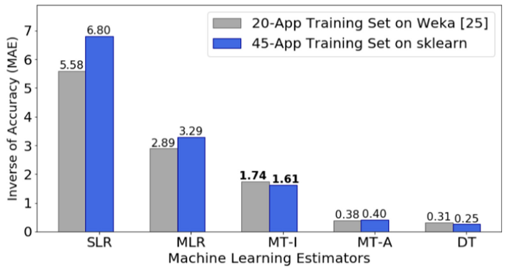
4.2 Results and Analysis: Model Training/Selection Phase

In this extension work, note that we focus on the improvements of the interpretable machine learning enhanced model training/selection methods: (1) consideration of the model interpretability by proposing the quantitative SOC, (2) flexible and powerful scikit-learn in Python-based machine learning techniques, and (3) effects on model robustness by using more data samples/instances.

In terms of model interpretability and accuracy using the two quantitative SOC and MAE metrics, the scikit-learn [31]-based more flexible and accurate model selection process can be achieved, compared to the ad hoc model structure categories [25]. As a result, we describe the improvements of model training/selection process from the perspective of interpretability and accuracy through the two steps: (1) Inter-model (among differerent estimators) evaluations in Figure 12 and 13 and (2) Intra-model (MT-model) evaluations in Table 21 and 22, with Figures 14 and 15).



(a) Δ FPS by ΔCPU-Freq



(b) Δ FPS by ΔGPU-Freq

Fig. 13. Inter-Model Evaluation Results on Accuracy (10-fold CV).

Interpretability (Inter-Model): As shown in Figure 12, we compare model interpretability between the ad hoc model structure categories of *simple, medium* and *complex* [25] and the quantitative simulatability metric (SOC). According to the ad hoc model structure, SLR, MLR, and MT-I can be categorized as the same *simple* models and MT-A and DT as the same *medium* models. Although the SLR could be the simplest model, to compare the interpretability between MLR and MT-I (or between MT-A and DT) is a very difficult case, not having a quantitative metric or threshold. However, if we use the SOC formulas (Table 2) using the required parameters (Table 3), then the #SOC of MLR (8) and that of MT-1 (13) can be clearly compared quantitatively. Moreover, by using a specific threshold (e.g., #SOC <= $\theta 1$, 15), the constrained SLR, MLR, and MT-I only can be candidate models. Last, MLP, SVR, and KNN among all the compared estimators (Table 1) are categorized as the *complex* structure (not interpretable models), which have the #SOCs (over 2500) in Figure 2 and were not included in the inter-model comparisons in Figure 12 and 13.

Accuracy (Inter-Model): As shown in Figure 13, we compare the model evaluation results on accuracy using the 10-fold **cross-validation (CV)** technique. The results of the chosen model (MT-I) in the 45-app dataset are almost similar or better (0.50 vs. 0.52 in Figure 13(a) and 1.74 vs. 1.61 in Figure 13(b)), compared to our 20-app dataset-based preliminary work [25]. In the effects of increased data samples/instances, which do not mean automatic increases of accuracy but depend on datasets and estimators. While the 45-app dataset has a little lower/better validation errors on average in the ΔCPU-Freq dataset (Figure 13(a)), the 20-app dataset has a little lower/better validation errors on average in the ΔGPU-Freq dataset (Figure 13(b)). In addition, the standard deviations in MAE between the 20-app and the 45-app datasets are also different in each estimator: SLR, MLR, MT-I, MT-A, DT (0.42, 0.22, **0.01**, 0.045, 0.04 in Figure 13(a) and 0.61, 0.2, 0.05, **0.01**, 0.03 in Figure 13(b)); and the MT-I (Figure 13(a)) and the MT-A (Figure 13(b)) have the lowest standard deviation values, representing MT models with appropriate max depths are less influential (robust) by the effects of the number of data samples/instances.

Furthermore, to have more improved intra-model (MT) evaluations using the same threshold (#SOC <= $\theta 1$, 15), the number of max depth (#D) and the number of features (#P) can be adjusted, according to the MT SOC formula ($2 \times D + 2 \times P + 1$), as shown in Table 21. While the #D is a hyper-parameter, the selection of the reduced #Ps requires additional evaluation process for feature selection/reduction. For this, we apply a greedy search, removing the least important features one by one, in terms of the MAE validation errors using 5- and 10-fold CV results using the *sklearn.model_selection. GridSearchCV* [31]. (The CV results between 5-fold and 10-fold are the same except only high #D (over 6D), shown in Table 21). Finally, the 3P (ΔF_C, Q and U_C) and the 2P (ΔF_C and U_C) are selected with the minimum MAE validation results, shown in Table 21. Based on the greedy search evaluation, we analyze the most important features (except ΔF_C) are

Table 21. Feature and Max Depth Configurations for Intra-model (MT) Selection Using Greedy Search Evaluation

#Max Depth (#D) #Features (#P)	1D	2D	3D	4D	5D	6D	10D	**Chosen Features**
4P ($\Delta F_C, Q, U_C, C_G$)	**0.66**	**0.52**	**0.41**	0.32	0.27	0.24 (0.23)	0.21	**Fixed**
3P ($\Delta F_C, Q, U_C$)	**0.66**	**0.52**	**0.40**	**0.33**	0.29	0.27 (0.26)	0.25 (0.24)	**Minimum (MAE)**
3P ($\Delta F_C, U_C, C_G$)	0.71	0.55	0.49	0.43	0.37	0.33	0.31 (0.30)	
3P ($\Delta F_C, Q, C_G$)	1.01	0.73	0.64	0.54	0.45	0.40	0.31 (0.30)	
2P ($\Delta F_C, U_C$)	**0.71**	**0.56**	**0.53**	**0.45**	**0.45**	0.44 (0.43)	0.45 (0.44)	**Minimum (MAE)**
2P ($\Delta F_C, Q$)	1.12	1.01	0.95	0.88	0.83	0.73	0.61 (0.59)	
2P ($\Delta F_C, C_G$)	1.20	1.17	1.15	1.11	1.02	0.96 (0.94)	0.82 (0.81)	
1P (ΔF_C)	**1.44**	**1.44**	**1.44**	**1.44**	**1.44**	**1.44**	1.44	**Fixed**

(The values in the brackets are the 10-fold CV errors, otherwise the same for 5- and 10-fold CVs)

Table 22. Intra-Model Evaluation Results of MT 1 on Interpretability, Δ FPS by ΔCPU-Freq: Using MT SOC Formula and $\theta 1$ (#SOC = $2 \times D + 2 \times P + 1 <= 15$)

#Max Depth (#D) #Features (#P)	1D	2D	3D	4D	5D	6D	10D	**Candidate Models** (#SOC <= $\theta 1$, 15)
4P ($\Delta F_C, Q, U_C$ and C_G)	**11**	**13**	**15**	17	19	21	29	4P: 1D,2D,3D
3P ($\Delta F_C, Q$ and U_C)	**9**	**11**	**13**	**15**	17	19	27	3P: 1D,2D,3D,4D
2P (ΔF_C and U_C)	**7**	**9**	**11**	**13**	**15**	17	25	2P: 1D,2D,3D,4D,5D
1P (ΔF_C)	**5**	**7**	**9**	**11**	**13**	**15**	23	1P: 1D,2D,3D,4D,5D,6D

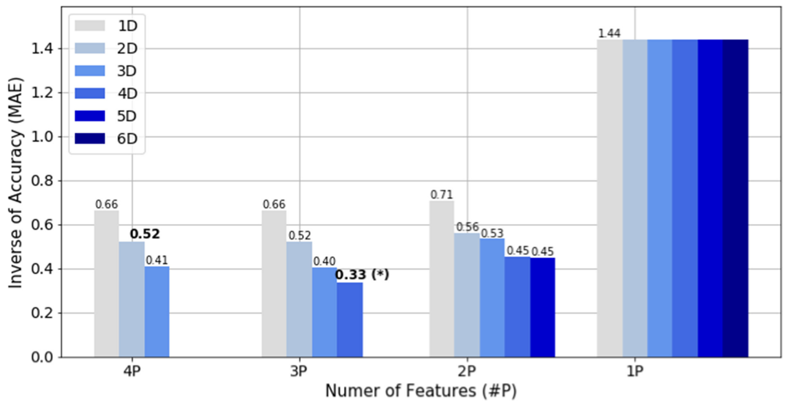


Fig. 14. Intra-model evaluation results of MT 1 on accuracy, Δ FPS by ΔCPU-Freq : Using 10-fold CV in MAE.

enumerated in order: U_C, Q and C_G; and C_G is the least important feature and slightly affects only in high #Max Depth (i.e., from #D of 4D).

Interpretability (MT Intra-Model): As shown in Table 22, using the same model selection process in Algorithm 1 with the help of the MT SOC formula and the threshold $\theta 1$, the #SOC can be easily calculated by the configurations of the number of max depth (#D) and the number of features (#P); and the threshold $\theta 1(15)$ determines the candidate model entries (highlighted in bold text).

Accuracy (MT Intra-Model): Among the candidate model entries, Figure 14 shows that the best model selection of MT 1 is the configuration of (3P, 4D) with the least cross-validation error of **0.33**($\star$), which is 36.54% improvement from the inter-model MT-I of **0.52** (4P, 2D), satisfying the other threshold $\theta 2$ (5% of 0.33 = 0.3465). In other words, as long as there is no more interpretable

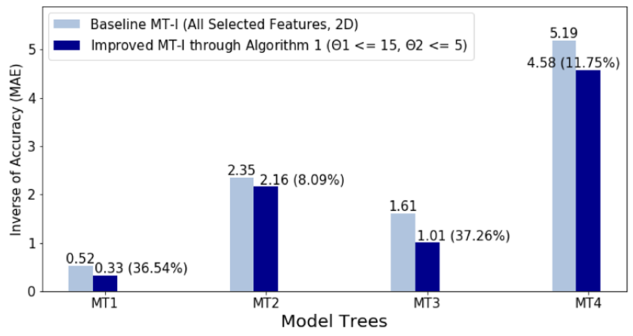


Fig. 15. Results of the improvements in MTs intra-model selection.

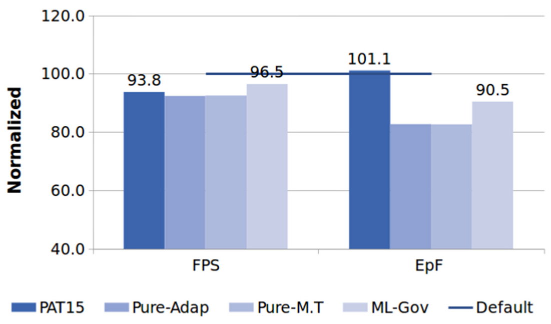


Fig. 16. Average results of the test set [25].

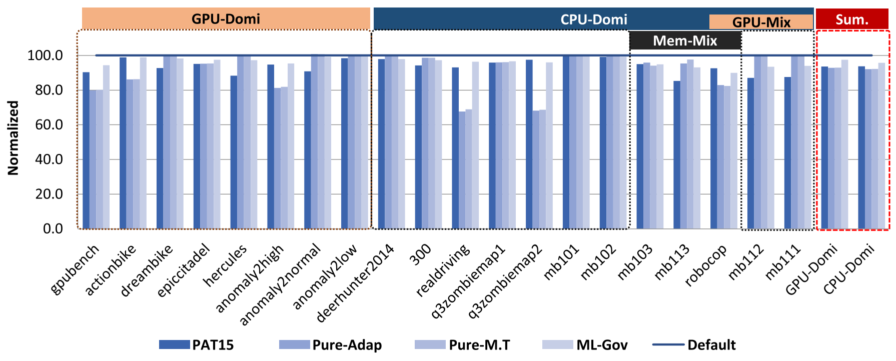
(less #SOC) candidate model having less than the MAE of 0.3465, the least CV error model can be the finally selected model according to the Algorithm 1.

Using the same MT intra-model selection process in terms of interpretability and accuracy with the Algorithm 1 and the two thresholds ($\theta 1 = 15$ and $\theta 2 = 5$), the improvements of all the MTs 1–4 in 10-fold CV errors are summarized, respectively, 36.54%, 8.09%, 37.26%, and 11.75%, as shown in Figure 15. In detail, in addition to the MT 1 improvement from 0.52 (4P, 2D) to 0.33 (3P, 4D), MT 3 from 1.61 (4P, 2D) to 1.01 (3P, 4D), in which the most important three features are in order: Q, U_C and U_G; and U_G is the least important feature only slightly affecting in improving accuracy. However, MT 2 from 2.35 (5P, 2D) to 2.16 (3P, 4D), in which the most important four features are in order: C_C, Q, C_G and U_G; but C_G and U_G affect almost similarly in accuracy so that it has a relatively lower improvement (8.09%) by removing the two least important features. MT 4 from 5.19 (2P, 2D) to 4.58 (2P, 5D), in which additional feature reduction is not possible in terms of accuracy; and it has a relatively lower improvement (11.75%) by the change of max depth up to 5D. Therefore, if the model evaluation results on model training/selection phase are satisfied and improved in terms of prediction CV errors, then we assume that the prediction phase could be inferred with improved prediction accuracy based on the satisfactorily improved built models.

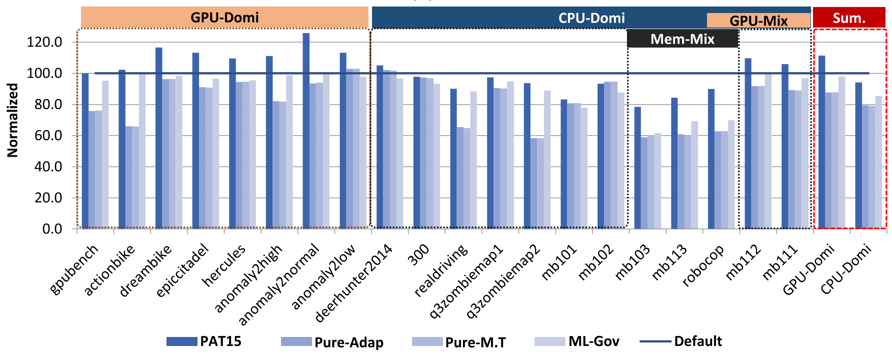
4.3 Results and Analysis: Prediction Phase

With the results and analyses on the model training/selection phase, the evaluation results of the test dataset on real platform are adopted like Figures 16 and 17 [25]. First, Figure 16 summarizes the average results of the test set in FPS and EpF on the real platform.

Our manager improves energy per frame by 9.5% and 10.6% on average compared to the default and PAT15 respectively, with minimal FPS decline of 3.5% compared to the default and 2.7% better



(a) FPS



(b) Energy per Frame

Fig. 17. Results of the test set (detailed) [25].

FPS than PAT15 on average. The two pure governors have 7.8% better EpF on average compared to our governor; but it results from the high FPS degradation of 7% compared to the default, while ours has 3.5% minimal FPS decline.

And then, Figure 17 shows the results of each application and summary of CPU- and GPU-dominant applications. First, Pure-Adap [28] and Pure-MT governors result in not-negligible FPS degradation up to 32% compared to the default in Figure 17(a), in some intensively CPU-dominant games such as *realdriving* (CPU Cost of 80–90 and GPU Cost of 40–60 in Figure 11) or *q3zombiemap2* (CPU Cost of 90–100 and GPU Cost of 20–40) games; in these cases we believe that the energy savings in Figure 17(b) can only be achieved with high-performance (FPS) degradation. Second, compared to PAT15 [29], Figure 17 clearly show that our tree-based piecewise linear regression model has significant impacts on energy savings with FPS improvement in the prediction phase, in addition to the improvements of the model accuracy (Table 8) in the learning phase. However, PAT15 using simple linear regression models has worse energy efficiency than the default governor as well as our governor; we observe that mainly GPU-dominant applications such as *Dreambike, Epicitadel*, and *Anomaly2* series (the first category in Figure 17) result in substantially worse EpF than the default, because the prediction errors of SLR by ΔF_G (Table 8) are significantly higher than those of SLR by ΔF_C.

Unlike GPU-dominant applications, most CPU-dominant applications for PAT15 achieved better EpF compared to the default (the second category in Figure 17). According to our observations, energy efficiency for mobile gaming benchmarks are mainly dependent on characteristics of benchmarks and platform characters such as CPU/GPU frequency levels and min/max frequency. (Note that the default CPU governor supports cluster-based interactive DVFS policy with nine frequency levels (from 1.2 to 2.0 GHz) in CPU-bc while the GPU governor supports six frequency levels from very low to high frequency (from 177 to 543 MHz)). From the observation that prediction errors of models to GPU frequency are higher than those to CPU frequency (Table 8), we speculate the reasons that all GPU cores are only dedicated for rendering tasks while CPU runs a lot of background tasks as well as the graphics rendering task running one of four cores and that response variables change sharply for GPU-dominant applications, because the ODROID-XU3 integrated GPU has small number of GPU frequency levels between min and max frequency (e.g., Intel MinnowBoard MAX integrated GPU has nine frequencies ranging from 200 to 511 MHz [11], compared to our six levels ranging from 177 to 543 MHz).

When GPU-workloads are mixed additionally onto CPU-dominant applications using our gaming micro-benchmarks [17] (e.g., *mb-101* and *mb102* vs. *mb111* and *mb112*: Each number stands for CPU, GPU, and Memory workloads, respectively), overall energy savings are reduced as GPU-workloads add on. Moreover, when memory-workloads are mixed onto CPU-dominant applications (e.g., *mb102* and *mb112* vs. *mb-103, mb113* and *robocop* game), all governors have EpF improvements without FPS degradation. This is because all compared governors except the default (only utilization-based policy) are using target-FPS–based policy. In other words, without a specific model that is aware of memory-workloads, a target-FPS–based policy can improve energy savings for memory intensive CPU-dominant applications, because QoS-based governors can repeatedly reduce the CPU frequency within the target-FPS.

5 CONCLUSION

In this article, we proposed *a machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming*. It exploits model building through offline machine learning techniques; and uses an integrated CPU-GPU DVFS methodology estimating energy-efficient frequencies with the models during runtime. For the learning phase, we collected training data using a set of 45 mobile games exhibiting diverse and dynamic characteristics; we performed attribute selection to remove unrelated variables to a response variable and reduce the complexity of structure of built models; and we built tree-based piecewise regression models using machine learning techniques built in the scikit-learn machine learning in Python. For the prediction phase, we developed a heuristic Hierarchical FSM-based governor framework considering QoS and CPU/GPU bottlenecks; we then set the CPU and GPU frequencies at runtime using the built models as outputs of the HFSM state transitions. Our experimental results on the ODROID-XU3 platform across a test set of 20 mobile games show that our governor achieved significant energy efficiency gains of over 10% (up to 38%) improvement in energy-per-frame over a state-of-the-art governor that built simple linear regression models, with a surprising 3% improvement in FPS performance. We believe our work presents a practical machine learning approached method to build models from dynamic data at numerous different hardware configurations on dynamic applications (workloads) of HMPSoC platforms. The work presented here uses offline machine learning techniques for online estimation, with future work addressing: a fair comparison between our supervised ML approach and recent prominent RL approaches in terms of different perspectives such as accuracy, interpretability, inference time or/and energy savings for integrated CPU-GPU DVFS governor. Finally, while our methodology was targeted mainly for mobile games, we believe it can also be applicable for various other classes of CPU-GPU integrated graphics applications.

REFERENCES

[1] Naomi S. Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *Am. Stat.* 46, 3 (1992), 175–185.

[2] Yu Bai and Priya Vaidya. 2009. Memory characterization to analyze and predict multimedia perormance and power in embedded systems. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'09)*.

[3] B. Bhattacharya and D. P. Solomatine. 2003. Neural networks and M5 model treesin modeling water level-discharge relationshipfor an Indian river. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN'03)*. 407–412.

[4] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. 1984. *Classification and Regression Trees*. Wadsworth Inc.

[5] Christoph Molnar. 2020. Interpretable Machine Learning: A Guide for Making Black Box Models Explainable. Retrieved from https://christophm.github.io/interpretable-ml-book/index.html.

[6] Po-Kai Chuan, Ya-Shu Chen, and Po-Hao Huang. 2017. An adaptive on-line cpu-gpu governor for games on mobile devices. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'17)*.

[7] Benedikt Dietrich and Samarjit Chakraborty. 2014. Lightweight graphics instrumentation for game state-specific power management in Android. In *Multimedia Systems*.

[8] Logan Dillard. 2017. lmt.py. Retrieved from https://gist.github.com/logandillard/lmt.py.

[9] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. 2012. A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC'12)*.

[10] Ujjwal Gupta, Manoj Babu, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, Suat Gumussoy, and Ümit Y. Ogras. 2018. An online learning methodology for performance modeling of graphics processors. *IEEE Trans. Comput.* 67, 12 (2018), 1677–1691.

[11] Ujjwal Gupta, Joseph Campbell, Raid Ayoub, Michael Kishinevsky, and Suat Gumussoy. 2016. Adaptive performance prediction for integrated GPUs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'16)*.

[12] Ujjwal Gupta, Sumit K. Mandal, Manqing Ma, Chaitali Chakrabarti, and Ümit Y. Ogras. 2019. A deep Q-learning approach for dynamic management of heterogeneous processors. *Comput. Arch. Lett.* 18, 1 (2019), 14–17.

[13] M. A. Hall. 1998. *Correlation-based Feature Subset Selection for Machine Learning*. Ph.D. Dissertation. University of Waikato, Hamilton, New Zealand.

[14] Jiawei Han and Micheline Kamber. 2001. *Data Mining Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA.

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2008. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

[16] Chen-Ying Hsieh, Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. 2018. Memcop: Memory-aware co-operative power management governor for mobile games. *Des. Autom. Embed. Syst.* 22 (Sep. 2018), 95–116.

[17] Jurn-Gyu Park. 2020. Gaming Applications (benchmarks) and Datasets. Retrieved from https://github.com/OD-ML/MLGov_ext.

[18] David Kadjo, Raid Ayoub, Michael Kishinevsky, and Paul V. Gratz. 2015. A control-theoretic approach for energy efficient CPU-GPU subsystem in mobile systems. In *Proceedings of the Design Automation Conference (DAC'15)*.

[19] Jaehwan Lee, Sanghyuck Nam, and Sangoh Park. 2019. Energy-efficient control of mobile processors based on long short-term memory. *Access IEEE* 7 (2019), 80552–80560.

[20] Zachary C. Lipton. 2018. The mythos of model interpretability. *Queue* 16, 3 (2018), 80552–80560.

[21] Sumit K. Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y. Ogras. 2019. Sumit K. Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, Umit Y. Ogras. *IEEE Trans. VLSI Syst.* 27, 12 (2019), 2842–2854.

[22] Michael Nielsen. 2019. Neural Networks and Deep Learning. Retrieved from https://neuralnetworksanddeeplearning.com/indexs.html.

[23] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. 2019. Quantifying Model Complexity via Functional Decomposition for Better Post-Hoc Interpretability. arXiv:1904.03867v2.

[24] W. James Murdocha, Chandan Singh, Karl Kumbiera, Reza Abbasi-Aslb, and Bin Yu. 2019. Interpretable machine learning: definitions,methods, and applications. arXiv.

[25] Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. 2017. ML-Gov: A machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming. In *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'17)*. 12–21.

[26] Jurn-Gyu Park, Chen-Ying Hsieh, Nikil Dutt, and Sung-Soo Lim. 2014. Quality-aware mobile graphics workload characterization for energy-efficient DVFS design. In *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'14)*.

[27] Jurn-Gyu Park, Chen-Ying Hsieh, Nikil Dutt, and Sung-Soo Lim. 2018. Synergistic CPU-GPU Frequency Capping for Energy-efficient Mobile Games. *IEEE Trans. Embed. Comput. Syst.* 17, 2 (2018).

[28] Jurn-Gyu Park, Hoyeonjiki Kim, Nikil Dutt, and Sung-Soo Lim. 2016. HiCAP: Hierarchical FSM-based dynamic integrated CPU-GPU frequency capping governor for energy-efficient mobile gaming. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'16).*

[29] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. 2015. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *Proceedings of the Design Automation Conference (DAC'15).*

[30] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. 2014. Integrated CPU-GPU Power Management for 3D Mobile Games. In *Proceedings of the Design Automation Conference (DAC'14).*

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[32] Ross J. Quinlan. 1992. Learning with Continuous Classes. In *Proceedings of the Australian Joint Conference on Artificial Intelligence.*

[33] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nat. Mach. Intell.* 1, 5 (2019), 206–215.

[34] Stuart Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

[35] Dylan Slack, Sorelle A. Friedler, Carlos Scheidegger, and Chitradeep Dutta Roy. 2019. Assessing the local interpretability of machine learning models. arXiv:1902.03501v2. Retrieved from https://arxiv.org/abs/1902.03501v2.

[36] Yong Wang and Ian H. Witten. 1997. Induction of model trees for predicting continuous classes. In *Proceedings of the European Conference on Machine Learning.*

[37] C. Willmott and K. Matsuura. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Clim. Res.* 30 (Dec. 2005), 79–82.

[38] Anson Wong. 2018. Building Model Trees. Retrieved from https://github.com/ankonzoid/LearningX/tree/master/advanced_ML/model_tree.

[39] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'15).*

Received December 2020; revised June 2021; accepted June 2021