ML-Gov: A Machine Learning Enhanced Integrated CPU-GPU DVFS Governor for Mobile Gaming

Jurn-Gyu Park, Nikil Dutt
School of Information and Computer Science
University of California, Irvine, CA, USA
jurngyup, dutt@ics.uci.edu

Sung-Soo Lim
School of Computer Science
Kookmin University, Seoul, South Korea
sslim@kookmin.ac.kr

ABSTRACT

Modern heterogeneous CPU-GPU based mobile architectures that execute intensive mobile games and other graphics applications use software governors to achieve high performance with energy-efficiency. For dynamic and diverse gaming workloads on heterogeneous platforms, existing governors typically utilize statistical or heuristic models assuming linear relationships for a small set of mobile games, resulting in high prediction errors. To overcome these limitations, we propose ML-Gov: a machine learning enhanced integrated CPU-GPU governor that builds tree-based piecewise linear models offline, and deploys these models for online estimation into an integrated CPU-GPU Dynamic Voltage Frequency Scaling (DVFS) 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% improvements on average in energy-per-frame with a surprising-but-modest 3% improvement in Frames-per-Second (FPS) performance, compared to a typical state-of-the-art governor that employs simple linear regression models.

CCS Concepts

•Computer systems organization → Embedded systems;

Keywords

Machine Learning techniques, Model-based design; Power management policies; DVFS; Integrated GPU

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 towards 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 [12] [9] for a small set of mobile games, assuming fairly static gaming workloads.

However, gaming applications exhibit inherent dynamism in their workloads, and recent research on software governors typically use classical statistical methods (e.g., simple linear regression models [12] 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 [15]. In order 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) [18] and High-Performance Computing (HPC) [5] have developed performance prediction models using machine learning techniques. But – to the best of our knowledge – machine learning based approaches have not been investigated for CPU-GPU integrated governors managing gaming workloads on mobile heterogeneous MPSoCs.

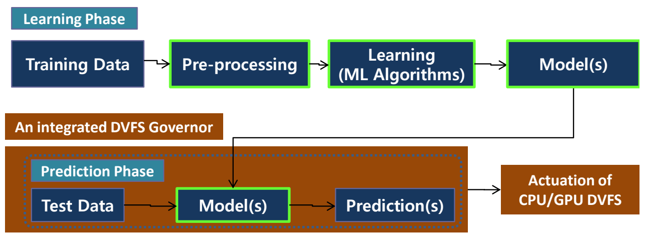


Figure 1: Machine Learning Approach for our System

To address these issues we propose ML-Gov: 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 piecewise linear models with high accuracy and a simple cost function structure 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

ESTIMedia'17, October 15–20, 2017, Seoul, Republic of Korea
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-5117-1/17/10. . . $15.00
DOI: https://doi.org/10.1145/3139315.3139317

of our governor is to achieve a target FPS or/and utilization as much as possible with minimal total power consumption using the integrated CPU-GPU DVFS.

Our paper makes the following specific contributions:

- We propose a practical machine learning approached tree-based piecewise regression model building methodology for diverse and dynamic gaming workloads on heterogeneous mobile platforms

- 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% 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 paper is organized as follows: Section 2 gives motivation and related work. Section 3 distinguishes our methodology using the learning and prediction phases. Section 4 shows and analyzes our results. Finally Section 5 concludes with a summary and future work.

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 (i.e., not just higher accuracy), and 2) a simple structure of cost functions 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 cost functions of the 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 (k-NN)) or neural networks that provide high accuracy, they also suffer from high opacity (not revealing anything about the structure of cost functions). On the other hand, the class of regression models provides simple cost functions but low accuracy.

To solve the challenge of predicting real values for non-linear type datasets, we deploy a tree-based piecewise linear model (i.e., model trees [15]), which combines a conventional decision tree with the possibility of linear regression functions at the leaves, that provides high accuracy with a simple cost function structure 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).

Therefore we found it important to deploy a simple and analyzable cost function structure 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 structure is simple due to a small number (3-5) of leaf nodes in a tree and the regression functions do not involve many attributes due to feature selection.

2.1 Motivation

Now consider Figure 2, where we compare model accuracy (prediction errors) and complexity of model cost functions among the machine learning algorithms (Table 1) to motivate the need for tree-based piecewise regression models. (For the motivating example in Figure 2, we use two datasets, which are collected from a real platform using a training set of 20 diverse games; and the characteristics of the dataset and platform configurations will be described in detail in Sections 3.1.1, 4.1, and Appendix B).

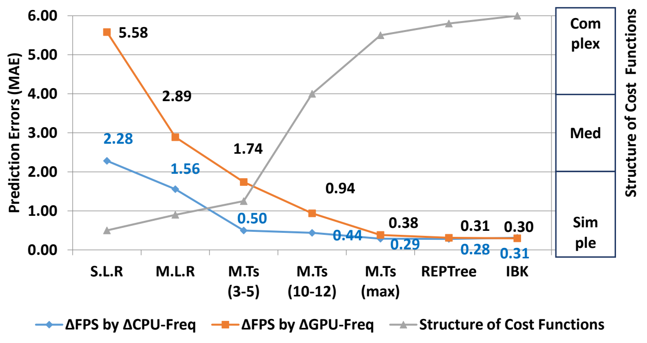


Figure 2: Comparison of Prediction Errors and Structure of Cost Functions among the Machine Learning Algorithms (Motivating Example)

Table 1: Compared M.L Algorithms

Algorithms in Weka [7]	Descriptions
S.L.R	Simple Linear Regression
M.L.R	Multivariate Linear Regression
M5P [14] [15]	Model Trees (M.Ts)
REPTree	Decision (Classification/Regression) Trees
IBk [1]	k-Nearest Neighbors (k-NN) [16]

The machine learning algorithms described in Table 1 are already built in a data-mining tool called *Weka* [7]. ΔFPS by Δ*CPU-Freq* (Δ*GPU-Freq*) in Figure 2 is FPS sensitivity to CPU (GPU) frequency change after feature selection. As a metric for prediction errors (the left y axis), we use Mean Absolute Error (MAE), which is a good indicator of average model performance [17]. The structure of cost functions (the right y axis) is compared relatively using simple, medium and complex levels among the algorithms. In other words, while an instance-based learning like k-NN does not reveal anything about the structure of the function (highest complexity) because of the non-parametric property [16], a simple linear regression provides the simplest structure. For the model trees, if the decision tree is simple (i.e., a small number of leaf nodes) and the regression functions do not normally involve many variables after a feature selection, the representation is simple and analyzable. However, as the number of leaf nodes in a tree increases, the complexity will also increase as shown in the middle of Figure 2.

For instance, while the average MAE of S.L.R and M.L.R for ΔFPS *by* ΔCPU-*Freq* (i.e., FPS sensitivity to CPU frequency change) is 2.28 and 1.56 respectively, that of M.T using 3-5 leaf nodes is 0.5 (i.e., the MAE of the M.T was reduced by 78% and 68% compared to S.L.R and M.L.R respectively). From the perspective of cost function complexity, M.Ts (3-5) after a feature selection have almost similar complexity with the multivariate linear models. This comparison clearly shows the need for a simple and analyzable structure of cost functions with high accuracy; model trees address this appropriately, with a small number of leaf nodes in a tree and analyzable 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 [13] 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 [12] 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.* [9] used a queuing model to capture CPU-GPU-Display interactions across a narrow range of games exhibiting limited diversity in CPU-GPU workloads. Our previous work proposed a coordinated CPU-GPU maximum frequency capping technique [10] and applied it to a diverse range of mobile graphics workloads, but assumed static characterization of each application; our next effort [11] 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 in order 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.* [5] proposed a method for estimating performance degradation on multicore processors and applied into HPC workloads; Wu *et al.* [18] 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 a simple cost function 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.

Gupta *et al.* [6] 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; 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.* [3] 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.

To the best of our knowledge, our work – unlike previous efforts focused purely on performance without regard to generality for mobile systems – is the first to introduce a practical machine-learning approached piecewise linear model building methodology that achieves high accuracy while using a simple cost function, 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 gaming workloads using a test set of 20 mobile games exhibiting diverse characteristics executing on a mobile HMPSoC platform.

3. ML-GOV METHODOLOGY

Our ML-Gov methodology has two phases: a learning phase and a prediction phase as shown in Figure 3.

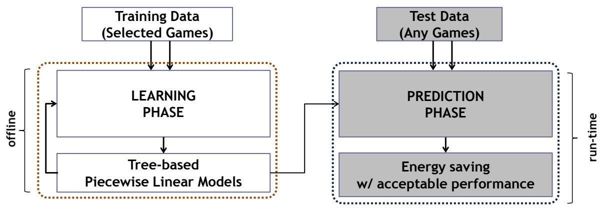

Training Data (Selected Games)
Test Data (Any Games)
offline
LEARNING PHASE
Tree-based Piecewise Linear Models
PREDICTION PHASE
Energy saving w/ acceptable performance
run-time

Figure 3: ML-Gov Overview

In the learning phase, we build tree-based piecewise linear regression models as well as comparable statistical models using offline machine learning techniques built in a data mining tool. Then in the prediction phase, ML-Gov 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 structure of cost functions 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 [12] [11].

3.1 Learning Phase

As shown in Figure 4, 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 complexity of cost functions and high accuracy to integrate the models easily into an integrated CPU-GPU DVFS governor.

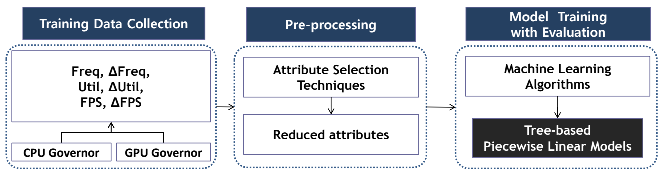
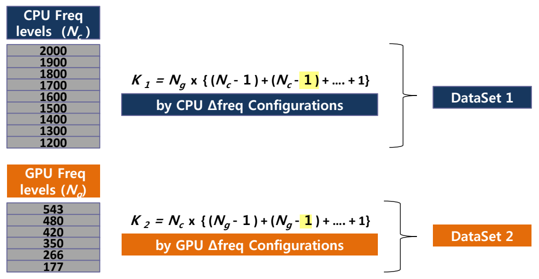

Training Data Collection
Freq, ΔFreq,
Util, ΔUtil,
FPS, ΔFPS
CPU Governor
GPU Governor
Pre-processing
Attribute Selection Techniques
Reduced attributes
Model Training with Evaluation
Machine Learning Algorithms
Tree-based Piecewise Linear Models

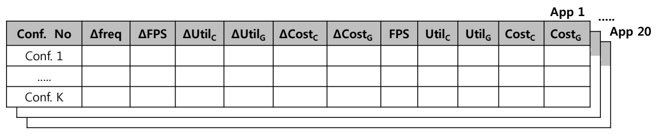
Figure 4: Learning Phase

3.1.1 *Collection of Training Data*

For training data, we use a training set of 20 mobile games 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] [13] [10] [11]; and we collect data at numerous different CPU and GPU frequency configurations using the diverse applications (workloads) on a mobile HMPSoC platform.


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

(a) Frequency Configurations



(b) DataSet

Figure 5: Data Collection Methodology

As shown in Figure 5.(a), 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 supported by the target system (9 frequency levels in CPU) at each GPU frequency of 6 frequency levels in GPU; and we measure another raw dataset by sweeping the GPU frequency with the same methodology. 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 5.(b)) based on comprehensive observations of related work [13] [12] [9] [10] [11] 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, we collect two datasets of 7020 = 20 x (36x6 + 15x9) instances with 351 different frequency configurations (216 for CPU and 135 for GPU frequency effects).

3.1.2 *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, and 2) to reduce the complexity of cost functions 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 5) and comparing their accuracy; the technique that achieved the lowest error rate was *correlation based feature subset attribute selection (CfsSubset)* [8] 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 2 shows the results of attribute selection.

Table 2: Selected Variables after Attribute Selection

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

3.1.3 *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 2, we mainly train the first three algorithms among the algorithms described in the motivating example: S.L.R, M.L.R and M.Ts (3-5). We note that instance-based learning does not reveal anything about the structure of cost functions due to the property of a non-parametric method; and that regression trees approximating a non-linear function by discretizing hundreds of leaves also are not adequate for our integrated governor. Furthermore, M.Ts (max) are 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 M.Ts (3-5) by changing *minNumInstances* (the minimum number of instances allowed at a leaf node) in the M5P algorithm. We empirically choose a model tree that has the smallest number among 3-5 leaf nodes, because it has a simple (analyzable) cost function structure while its prediction errors are significantly lower than those of M.L.R or the model trees having only 2 leaf nodes.

Table 3 summarizes the results of our model evaluation. The last column of this table shows our final selection, comprising four M.Ts (1-4) and two S.L.Rs (1-2) for the six responsive variables. These selections were made based on their lower prediction errors and simple cost function struc-

Table 3: Prediction Errors for Model Evaluation

Response Var.	S.L.R	M.L.R	C-M.T	F-M.T	Chosen
ΔQ by ΔF_C	3.79%	2.58%	0.83% (3)	0.48%	M.T 1
ΔU_C by ΔF_C	15.55%	12.73%	11.15% (5)	6.33%	M.T 2
ΔU_G by ΔQ_{F_C}	4.49%	4.49%	4.44% (5)	4.46%	S.L.R 1
ΔQ by ΔF_G	9.96%	5.15%	3.10% (4)	0.68%	M.T 3
ΔU_G by ΔF_G	19.82%	15.24%	14.04% (4)	8.42%	M.T 4
ΔU_C by ΔQ_{F_G}	22.78%	22.71%	22.55% (5)	16.59%	S.L.R 2

tures: Model prediction errors for ΔU_G by ΔQ_{F_C} and ΔU_C by ΔQ_{F_G} are almost similar for all algorithms so that we use the simplest S.L.R models However, for the other response variables, we use the M.Ts (1-4) because of high accuracy and analyzable cost functions.

We illustrate the structure of the Model Trees using M.T 1 as an example (the other model trees – M.T 2, 3 and 4 – use the same model building methodology and are presented in the Appendix for completeness); for the regression coefficients, we use α, β, γ, θ and λ respectively for each regression function (i.e., LM#) with the numbering for the different coefficients. Model Tree M.T 1 is a tree-based piecewise linear regression model to predict ΔQ by ΔF_C. Lines 1-4 reveal a

Model Tree M.T 1: ΔQ by ΔF_C

1: $U_C <= 82$: LM1 (2774/16.283%)
2: $U_C > 82$:
3: | $Q <= 54$: LM2 (1160/57.601%)
4: | $Q > 54$: LM3 (386/20.231%)

▷ **Smoothed Mode**
5: LM1: ΔQ = - 0.0007 * Q + 0.0003 * U_C - 0.001 * C_G + 0.0603
6: LM2: ΔQ = 0.016 * ΔF_C - 0.0042 * Q + 0.0005 * U_C + 0.4079
7: LM3: ΔQ = 0.0004 * ΔF_C - 0.0098 * Q + 0.0005 * U_C + 0.8146

▷ **Un-smoothed Mode**
8: LM1: ΔQ = -0.001 * C_G + 0.0407
(we set -0.001=α_1^{MT1}, 0.0407=α_2^{MT1})
9: LM2: ΔQ = 0.0162 * ΔF_C + 0.2358
(we set 0.0162=β_1^{MT1}, 0.2358=β_2^{MT1})
10: LM3: ΔQ = + 0.3811
(we set 0.3811=γ_1^{MT1})

tree structure with the thresholds of U_C and Q, and Lines 5-7 and 8-10 correspond to each regression model in each leaf node with the parameters in smoothed and un-smoothed modes. In our model, we use un-smoothed mode instead of smoothed mode because the structure of the un-smoothed mode is simpler while the prediction errors between the two modes are exactly the same in our datasets.

And, the S.L.Rs for the two responsive variables (ΔU_G by ΔQ_{F_C} and ΔU_C by ΔQ_{F_G}) are as follows.

S.L.R 1: ΔU_G by ΔQ_{F_C}

ΔU_G = 0.8704 * ΔQ (we set 0.8704 = α_1^{SLR1})

S.L.R 2: ΔU_C by ΔQ_{F_G}

ΔU_C = 1.4575 * ΔQ (we set 1.4575 = α_1^{SLR2})

Model Equations: We derive model equations from the four model trees and the two S.L.Rs. 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.

To estimate the FPS at a higher 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 in Appendix) as follows (ΔF_C equals F'_C - F_C, and ΔF_G equals F'_G - F_G).

$$Q^{(F'_C,F_G)} - Q^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT1} C_G + \alpha_2^{MT1} & \text{if } LM1. \\ \beta_1^{MT1} \Delta F_C + \beta_2^{MT1} & \text{if } LM2. \\ \gamma_1^{MT1} & \text{if } LM3. \end{cases} \tag{1}$$

$$Q^{(F_C,F'_G)} - Q^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT3} & \text{if } LM1. \\ \beta_1^{MT3} & \text{if } LM2. \\ \gamma_1^{MT3} & \text{if } LM3. \\ \theta_1^{MT3} \Delta F_G + \theta_2^{MT3} Q + \theta_3^{MT3} & \\ & \text{if } LM4. \end{cases} \tag{2}$$

To estimate the utilization of a component at a higher 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) in the Appendix.

$$U_C^{(F'_C,F_G)} - U_C^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT2} \Delta F_C + \alpha_2^{MT2} Q + \alpha_3^{MT2} C_C + \alpha_4^{MT2} & \\ & \text{if } LM1. \\ \beta_1^{MT2} \Delta F_C + \beta_2^{MT2} C_C + \beta_3^{MT2} C_G + \beta_4^{MT2} & \\ & \text{if } LM2. \\ \gamma_1^{MT2} & \text{if } LM3. \\ \theta_1^{MT2} & \text{if } LM4. \\ \lambda_1^{MT2} U_G + \lambda_2^{MT2} C_G + \lambda_3^{MT2} & \text{if } LM5. \end{cases} \tag{3}$$

$$U_G^{(F_C,F'_G)} - U_G^{(F_C,F_G)} = \begin{cases} \alpha_1^{MT4} & \text{if } LM1. \\ \beta_1^{MT4} & \text{if } LM2. \\ \gamma_1^{MT4} & \text{if } LM3. \\ \theta_1^{MT4} \Delta F_G + \theta_2 & \text{if } LM4. \end{cases} \tag{4}$$

Finally, we estimate the impact of cross-component frequency variations on utilizations, which occurs as long as there is parallel increase in FPS [12]. Therefore, the corresponding equations can be derived as follows, by using the S.L.R2 (ΔU_C by ΔQ_{F_G}) and S.L.R1 (ΔU_G by ΔQ_{F_C}).

$$U_C^{(F_C,F'_G)} - U_C^{(F_C,F_G)} = \alpha_1^{S.L.R1}(Q^{(F_C,F'_G)} - Q^{(F_C,F_G)}) \tag{5}$$

$$U_G^{(F'_C,F_G)} - U_G^{(F_C,F_G)} = \alpha_1^{S.L.R1}(Q^{(F'_C,F_G)} - Q^{(F_C,F_G)}) \tag{6}$$

3.2 Prediction Phase

As shown in Figure 6, 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.

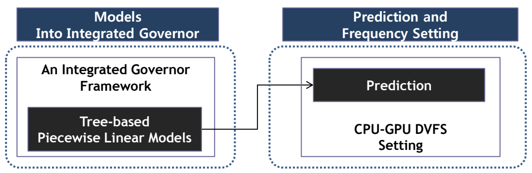

Models Into Integrated Governor
An Integrated Governor Framework
Tree-based Piecewise Linear Models
Prediction and Frequency Setting
Prediction
CPU-GPU DVFS Setting

Figure 6: Prediction Phase

3.2.1 An Integrated Governor Framework

We build our governor framework on the non-trivial observations gleaned from the related work [12] [10] [11]: 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 [12]), 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 [4] or CPU-GPU utilization patterns [13]. This kind of sampling method enables us to avoid prior comprehensive offline profiling [12], especially to compare with the default separate CPU and GPU governors (i.e., performance-driven policy without CPU-GPU cooperation).

Figure 7 illustrates our power management algorithm using the Hierarchical FSM-based representation [11], since it provides a natural and intuitive design abstraction.

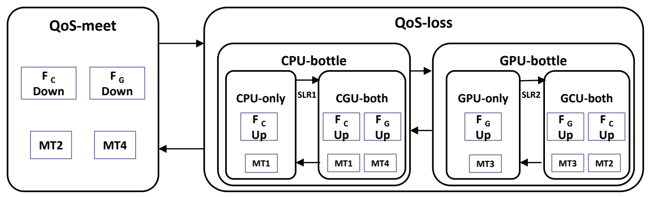


Figure 7: HFSM-based Power Management Algorithm

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 M.T 2 and 4). For *QoS-loss* super-state, there are two sub-states: if CPU is the bottleneck, *CPU-bottle* or if GPU is the bottleneck, *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 M.T 1); for *CGU-both* leaf state, GPU frequency will be scaled up to the estimated frequency (using M.T 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 M.T 3); for *GCU-both* leaf state, CPU frequency will be scaled up to the estimated frequency (using M.T 2) in addition to the GPU frequency scale up.)

3.2.2 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, 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.

$$F'_C = \begin{cases} \frac{(\hat{Q} - Q^{(F_C,F_G)}) - \beta_2^{MT1}}{\beta_1^{MT1}} + F_C & \text{if } LM = 2. \\ F_C{+}{+} & \text{otherwise.} \end{cases} \tag{7}$$

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 $\hat{U}_G$, 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).

$$F'_G = \begin{cases} \frac{(U_G^{(F_C,F_G)} - \hat{U}_G) - \theta_2^{MT4}}{\theta_1^{MT4}} + F_G & \text{if } LM = 4. \\ F_G{+}{+} & \text{otherwise.} \end{cases} \tag{9}$$

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 S.L.R 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).

$$F''_C = \begin{cases} \frac{(\hat{U}_C - U_C^{(F_C,F_G)}) - (\alpha_2^{MT2} * Q + \alpha_3^{MT2} * C_C + \alpha_4^{MT2})}{\alpha_1^{MT2}} + F_C & \\ & \text{if } LM = 1. \\ \frac{(\hat{U}_C - U_C^{(F_C,F_G)}) - (\beta_2^{MT2} * C_C + \beta_3^{MT2} * C_G + \beta_4^{MT2})}{\beta_1^{MT2}} + F_C & \\ & \text{if } LM = 2. \\ F_C{+}{+} & \text{otherwise.} \end{cases} \tag{10}$$

$$F''_G = \begin{cases} \frac{(\hat{U}_G - U_G^{(F_C,F_G)}) - \theta_2^{MT4}}{\theta_1^{MT4}} + F_G & \text{if } LM = 4. \\ F_G{+}{+} & \text{otherwise.} \end{cases} \tag{11}$$

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

We evaluated our ML-Gov manager on the ODROID-XU3 development board installed with Android 4.4.2 and Linux 3.10.9; Table 4 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 1.2Ghz to 2.0Ghz) in CPU-bc and at seven frequency levels (from 1.0Ghz to 1.6Ghz) in CPU-lc, and GPU supports six frequency levels (from 177Mhz to 543Mhz).

Table 4: Platform Configuration

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

Benchmark Set: We use a training set of 20 games and a test set of 20 other games including several micro-benchmarks, covering different types of graphics workloads. Appendix B summarizes the 20 training games and the 20 test set games.

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 energy per frame (EpF) for three runs (120 seconds for each run) of each benchmark, and averaged their measurements.

For comparison: We compare our ML-gov manager against the default, a state-of-the-art governor [12], adaptive only and model-tree only policies as shown in Table 5.

Table 5: Governors for Comparison

Governor	Description
Default	Interactive CPU and proprietary GPU governor
PAT15	Simple Linear Regression-based Governor
ML-Gov	Our Tree-based piecewise Linear Regression
Naive-Adap.	Naive one step higher/lower Adaptive policy
Naive-M.T	Only Tree-based Linear Models w.o Adaptation

The default corresponds to the independent CPU and GPU governors in Linux (Interactive CPU governor and ARM's Mali Midgard GPU governor). The state-of-the-art governor [12] (represented as PAT15) proposed an integrated CPU-GPU DVFS strategy by developing predictive simple linear regression power-performance models. The naive adaptive governor is represented as Naive-Adap in the figures which does not use the built model-trees but adaptively scale up to one step higher (or down to one step lower) frequency for corresponding component according to each leaf state. The naive model-tree governor is represented as Naive-M.T, which does not use the heuristic adaptation (i.e., 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, 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 20us, which is totally negligible compared to the epoch period (200ms) in terms of performance (FPS degradation). Moreover, we did not observe any noticeable increase in average power consumption due to our power manager. And, the reason we use the epoch of 200ms (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.

4.2 Results and Analysis

Figure 8 summarizes the average results of the test set in FPS and EpF. Our ML-Gov 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 FPS than PAT15 on average. The two naive governors have 7.8% better EpF on average compared to our governor, but it results from the high FPS degradation of 7% (up to 32% shown in Figure 9.(a)) compared to the Default; in these cases we believe that these energy savings can only be achieved with some degradation in performance.

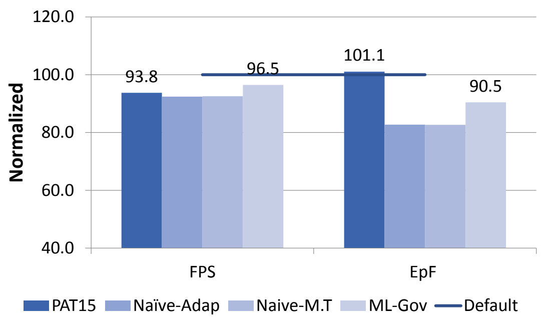
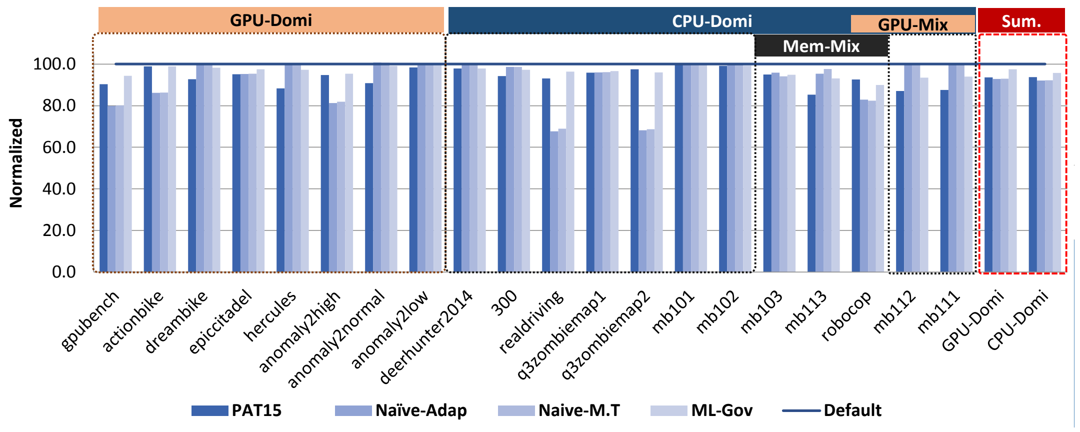


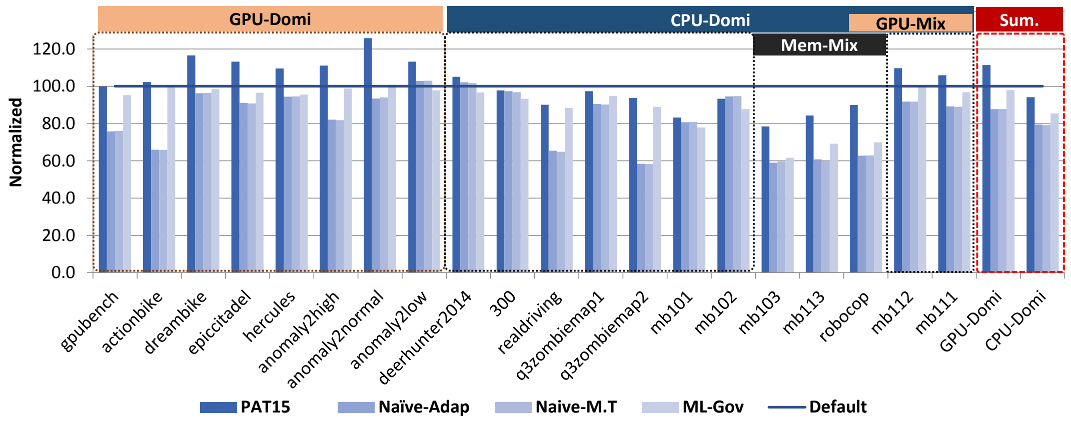
Figure 8: Average Results of the Test Set

Figure 9 shows the results of each application and summary of CPU- and GPU-dominant applications. Especially, compared to PAT15, Figure 8 and 9 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 3) in the learning phase. On the other hand, 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 9) result in substantially worse EpF than the default because the prediction errors of S.L.R by ΔF_G (Table 3) are significantly higher than those of S.L.R 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 9). According to our observations, energy efficiency for mobile gaming benchmarks are mainly dependent on characteristics of bench-



(a) FPS



(b) Energy per Frame

Figure 9: Results of the Test Set (Detailed)

marks 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 2.0Ghz) in CPU-bc while the GPU governor supports six frequency levels from very low to high frequency (from 177 to 543Mhz)). From the observation that prediction errors of models to GPU frequency are higher than those to CPU frequency (Table 3), 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 511Mhz [6], compared to our six levels ranging from 177 to 543Mhz).

When GPU-workloads are mixed additionally onto CPU-dominant applications using our gaming micro-benchmarks (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 paper, we proposed *ML-Gov: A Machine Learning Enhanced Integrated CPU-GPU DVFS Governor for Mobile Gaming*. ML-Gov 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 20 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 cost functions; and we built tree-based piecewise regression models using machine learning techniques built in the data mining tool. 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 run-time 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% improvement in energy-per-frame over a state-of-the-art governor which built simple linear regression models, with a surprising 3% improvement in FPS performance. We believe ML-Gov 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: online machine learning methodology and implementation for integrated CPU-GPU DVFS governor. Finally, while our ML-Gov methodology was targeted mainly for mobile games, we believe it can also be applicable for various other classes of CPU-GPU integrated graphics applications.

6. REFERENCES

[1] D. Aha and D. Kibler. Instance-based learning algorithms. In *Machine Learning*, 1991.

[2] Y. Bai and P. Vaidya. Memory characterization to analyze and predict multimedia performance and power in embedded systems. In *ICASSP*, 2009.

[3] P.-K. Chuan, Y.-S. Chen, and P.-H. Huang. An adaptive on-line cpu-gpu governor for games on mobile devices. In *Design Automation Conference (ASP-DAC)*, Jan. 2017.

[4] B. Dietrich and S. Chakraborty. Lightweight graphics instrumentation for game state-specific power management in android. In *Multimedia Systems*, 2014.

[5] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[6] U. Gupta, J. Campbell, R. Ayoub, M. Kishinevsky, and S. Gumussoy. Adaptive performance prediction for integrated gpus. In *ICCAD*, 2016.

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, June 2009.

[8] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning.* PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[9] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz. A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile systems. In *DAC*, 2015.

[10] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Co-cap: Energy-efficient cooperative cpu-gpu frequency capping for mobile games. In *SAC*, 2016.

[11] J.-G. Park, H. Kim, N. Dutt, and S.-S. Lim. Hicap: Hierarchical fsm-based dynamic integrated cpu-gpu frequency capping governor for energy-efficient mobile gaming. In *ISLPED*, Aug. 2016.

[12] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *DAC*, 2015.

[13] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU power management for 3D mobile games. In *DAC*, June 2014.

[14] R. J. Quinlan. Learning with continuous classes. In *Australian Joint Conference on Artificial Intelligence*, 1992.

[15] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *European Conference on Machine Learning*, 1997.

[16] Wikipedia. k-nearest neighbors algorithm. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

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

[18] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *HPCA*, 2015.

APPENDIX

A. MODEL TREES

The Model Tree 2 is a tree-based piecewise linear regression model to predict ΔU_C by ΔF_C. Line 1-8 reveals a tree structure with 5 leaf nodes by the thresholds of C_C, C_G and ΔF_C, (if ΔF_C or ΔF_G is included in the thresholds, we estimate it using another multivariate linear regression model); and Line 9-13 corresponds to each regression model in each leaf node with the parameters in the un-smoothed mode.

Model Tree 2: ΔU_C by ΔF_C

```
1: $C_C <= 61.152$
2: | $\Delta F_C <= 250$ : LM1 (1293/54.063%)
3: | $\Delta F_C > 250$ :
4: | | $C_C <= 54.416$ : LM2 (1549/52.608%)
5: | | $C_C > 54.416$ :
6: | | | $C_G <= 29.714$ : LM3 (162/9.39%)
7: | | | $C_G > 29.714$ : LM4 (243/198.57%)
8: $C_C > 61.152$ : LM5 (1073/22.061%)

9: LM1: $\Delta U_C$ = -0.018 * ΔFc - 0.0585 * Q - 0.0294 * $C_C$ + 4.1886
                  (we set -0.018=$\alpha_1^{MT2}$, - 0.0585=$\alpha_2^{MT2}$
                        - 0.0294=$\alpha_3^{MT2}$, 4.1886=$\alpha_4^{MT2}$)
10: LM2: $\Delta U_C$ = -0.0186 * ΔFc - 0.1521 * $C_C$ - 0.0415 * $C_G$ + 7.4911
                  (we set -0.0186=$\beta_1^{MT2}$, - 0.1521=$\beta_2^{MT2}$
                        - 0.0415=$\beta_3^{MT2}$, 7.4911=$\beta_4^{MT2}$)
11: LM3: $\Delta U_C$ = -0.1934
                  (we set -0.1934=$\gamma_1^{MT2}$)
12: LM4: $\Delta U_C$ = -14.7754
                  (we set -14.7754=$\theta_1^{MT2}$)
13: LM5: $\Delta U_C$ = -0.0078 * $U_G$ - 0.0534 * $C_G$ + 1.5904
               (we set -0.0078=$\lambda_1^{MT2}$, - 0.0534=$\lambda_2^{MT2}$, 1.5904=$\lambda_3^{MT2}$)
```

The Model Tree 3 is a tree-based piecewise linear regression model to predict ΔQ by ΔF_G.

Model Tree 3: ΔQ by ΔF_G

```
1: $U_G <= 96.86$ :
2: | $U_G <= 39.608$ : LM1 (171/3.352%)
3: | $U_G > 39.608$ :
4: | | $U_C <= 96.032$ : LM2 (395/12.61%)
5: | | $U_C > 96.032$ : LM3 (212/12.486%)
6: $U_G > 96.86$ : LM4 (933/60.453%)

7: LM1: $\Delta Q$ = + 0.0282
                  (we set 0.0282=$\alpha_1^{MT3}$)
8: LM2: $\Delta Q$ = + 0.7848
                  (we set 0.7848=$\beta_1^{MT3}$)
9: LM3: $\Delta Q$ = - 0.4865
                  (we set -0.4865=$\gamma_1^{MT3}$)
10: LM4: $\Delta Q$ = 0.031 * $\Delta F_G$ - 0.55 * $Q$ + 29.9607
               (we set 0.031=$\theta_1^{MT3}$, - 0.55=$\theta_2^{MT3}$, 29.9607=$\theta_3^{MT3}$)
```

The Model Tree 4 is a tree-based piecewise linear regression model to predict ΔU_G by ΔF_G.

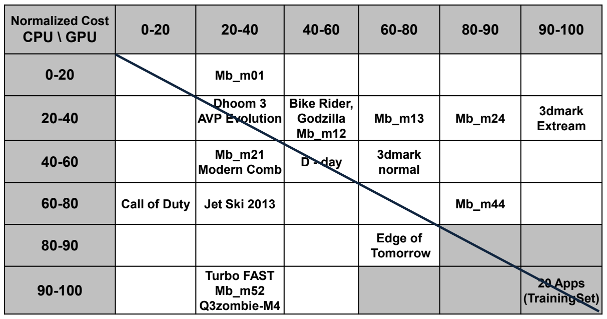
Model Tree 4: ΔU_G by ΔF_G

```
1: $\Delta F_G$ <= 142 :
2: | $\Delta F_G$ <= 66.5 : LM1 (360/13.196%)
3: | $\Delta F_G$ > 66.5 :
4: | | $U_G$ <= 99.148 : LM2 (704/42.046%)
5: | | $U_G$ > 99.148 : LM3 (196/25.485%)
6: $\Delta F_G$ > 142 : LM4 (1440/88.951%)

7: LM1: $\Delta U_G$ = - 3.8548
        (we set - 3.8548=$\alpha_1^{MT4}$)
8: LM2: $\Delta U_G$ = - 10.8804
        (we set - 10.8804=$\beta_1^{MT4}$)
9: LM3: $\Delta U_G$ = - 2.429
        (we set - 2.429=$\gamma_1^{MT4}$)
10: LM4: $\Delta U_G$ = -0.0896 * $\Delta F_G$ - 1.7533
        (we set -0.0896=$\theta_1^{MT4}$, - 1.7533=$\theta_2^{MT4}$)
```

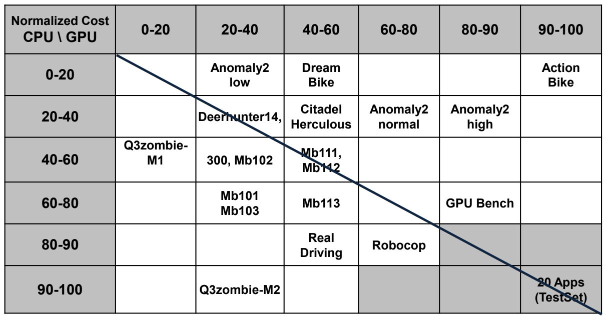
B. MOBILE GAMES: TRAINING SET AND TEST SET

Figures 10 and 11 summarize the 20 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 in order to cover the entire space of different graphics workloads.



Normalized Cost CPU \ GPU	0-20	20-40	40-60	60-80	80-90	90-100
0-20		Mb_m01				
20-40		Dhoom 3 AVP Evolution	Bike Rider, Godzilla Mb_m12	Mb_m13	Mb_m24	3dmark Extream
40-60		Mb_m21 Modern Comb	D - day	3dmark normal		
60-80	Call of Duty	Jet Ski 2013			Mb_m44	
80-90				Edge of Tomorrow		
90-100		Turbo FAST Mb_m52 Q3zombie-M4				20 Apps (TrainingSet)

Figure 10: The 20-App Training Set



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)

Figure 11: The 20-App Test Set

The CPU-GPU graphics workload varation and their relative intensity is quantified using a CPU-GPU cost metric that is a product of the utilization and frequency [2] [13] [10], as shown by the rows and columns of the tables in Figures 10 and 11.

We note that the gaming applications 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 CPU or GPU workloads, with the highest (e.g., 90-100 (level 5)) representing the most CPU- or GPU- intensive gaming applications or benchmarks.

In this context, the 20 test set gaming applications analyzed in Figure 9 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%).