Adaptive Performance Prediction for Integrated GPUs

Ujjwal Gupta, Joseph Campbell, Raid Ayoub, Michael Kishinevsky, Suat Gumussoy
Umit Y. Ogras Francesco Paterna IEEE Member
Arizona State University Intel Corporation

ABSTRACT

Integrated GPUs have become an indispensable component of mobile processors due to the increasing popularity of graphics applications. The GPU frequency is a key factor both in application throughput and mobile processor power consumption under graphics workloads. Therefore, dynamic power management algorithms have to assess the performance sensitivity to the GPU frequency accurately. Since the impact of the GPU frequency on performance varies rapidly over time, there is a need for online performance models that can adapt to varying workloads. This paper presents a light-weight adaptive runtime performance model that predicts the frame processing time. We use this model to estimate the frame time sensitivity to the GPU frequency. Our experiments on a mobile platform running common GPU benchmarks show that the mean absolute percentage error in frame time and frame time sensitivity prediction are 3.8% and 3.9%, respectively.

1. INTRODUCTION

Graphically-intensive mobile applications, such as games, are now one of the most popular smartphone application categories. There are more than a quarter million games, which led to several million downloads on Android devices alone [1]. When running many of these applications, the GPU power consumption accounts for more than 35% of application processor power. It is not always viable to decrease the GPU frequency to reduce power consumption, since graphics performance is highly sensitive to the frequency. Therefore, there is a need for accurate performance models that can be used to control the GPU frequency judiciously.

The primary GPU performance metric is the number of frames that can be processed per second, since this number determines the display frame rate. Therefore, we use the time the GPU takes to process a frame as the performance metric. The frame time varies significantly for different time periods of an application, as shown in Figure 1. Furthermore, it is highly correlated with the GPU frequency and dependent on the target application. Hence, the frame time is a multivariate function of the frequency and workload, where the latter is captured by the performance counters. An accurate GPU performance model can enable us to predict the *change in performance as a function of change in frequency*. The performance model needs to capture the impact of dynamic workload variations, and have a low computational overhead. Combined with a power model, this sensitivity model can be integrated into dynamic power management algorithms to select the best GPU frequency.

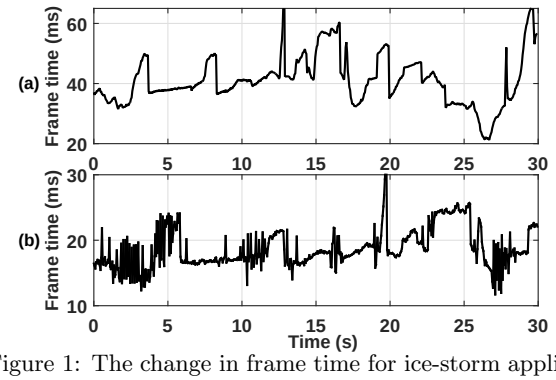

Figure 1: The change in frame time for ice-storm application for (a) 200 MHz and (b) 489 MHz GPU frequencies.

This paper presents a systematic methodology for constructing a tractable runtime model for GPU frame time prediction. The proposed methodology consists of two major steps. The first step is an extensive analysis to collect frame time and GPU performance counter data. This analysis enables us to construct a frame time model template and select the most significant feature set. Our model employs differential calculus to express the change in frame time as a function of the partial derivatives of the frame time with respect to the GPU frequency and performance counters. The second step is an adaptive algorithm that learns the coefficients of the proposed model online. More specifically, we employ a light-weight recursive least squares (RLS) algorithm to predict the change in frame time dynamically. RLS is a good choice since the correlation between different frames decays quickly unlike the fractal behavior observed at the macroblock level [19]. Besides frame time prediction, we use our model to predict the frequency sensitivity, which is defined as the derivative of the frame time with respect to the GPU frequency. This information can be utilized by dynamic power management algorithms, which often have to make a decision to change the frequency [2, 17]. To validate our approach, we performed experiments on a state-of-the-art mobile platform using both custom applications and commonly used graphics benchmarks [8]. The experiments show that the mean absolute percentage error in frame time and frame time sensitivity prediction are 3.8% and 3.9%, respectively.

The major contributions of this work are:

- A methodology for collecting offline data and developing an adaptive GPU performance model,
- A concrete RLS based adaptive runtime performance model,
- Extensive evaluations on a commercial platform using common GPU benchmarks.

ICCAD '16, November 07-10, 2016, Austin, TX, USA
 ISBN 978-1-4503-4466-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2966986.2966997

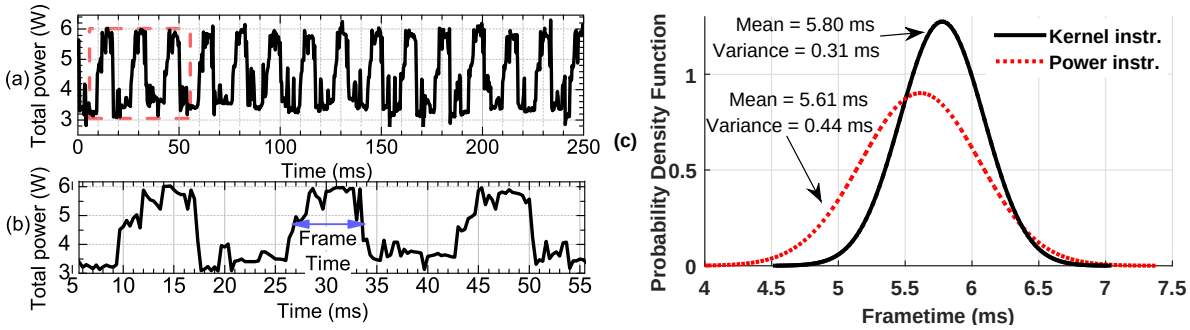

Figure 2: (a) Total power consumption when the GPU is rendering Art3 application at 60 FPS. (b) Zoomed portion, which shows three frames in the first 50ms. (c) Frame time distribution for kernel and power instrumentations for Art3 application.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 details the challenges and lays out the groundwork required for frame time prediction. Section 4 presents the techniques for offline analysis and online learning. Finally, Section 5 discusses the experimental results, and Section 6 concludes the paper.

2. RELATED RESEARCH

Dynamic power management techniques require accurate performance models to decide when and how much the frequency can be slowed down to save power. Therefore, this work focuses on light-weight performance models that can guide DPM algorithms in conjunction with runtime power models [12, 16]. The authors in [3] proposed a framework to optimize the CPU-GPU efficiency by classifying the application phases as rendering and loading. The GPU frequency governor boosts the GPU frequency during the rendering phase to improve performance, while reducing the frequency during the loading phase to reduce power consumption. A more direct approach to govern GPU performance based on the CPU, GPU frequencies and utilizations is presented in [17]. However, this model relies on utilization, instead of using the performance counters which provide a fine-grain measure of the workload. Kadjo et al. [14] use the performance counters, but learn the models using offline data only. Another work on performance modeling [4] uses an auto-regressive (AR) model for frame time prediction. The authors employ a tenth order AR model, whose weights were learned offline using ten minutes of frame time data for each application. Similarly, the authors in [5] utilize the Least Mean Squares estimation technique to predict graphics workloads with a model whose features are based on prior frame times. On the one hand, relying solely on offline data does not generalize well to other data sets, as it is not feasible to account for all possible workloads. On the other hand, online learning is challenging due to limited observability and computing resources. We address these concerns by providing an efficient technique for GPU performance prediction.

3. FRAME TIME CHARACTERIZATION

3.1 Challenges and Notation

The first step towards constructing a high fidelity frame time model is to understand the dependence of the frame time on the GPU frequency and workload. As mentioned before, the workload characteristics are captured by the performance counters $\mathbf{x} = [x_1, x_2, \ldots, x_N]$, where N is the total number of counters. All of these counters are functions of the frame complexity C, while some of them also depend on the GPU frequency f. Since the frequency changes in discrete time steps in practical systems, we characterize the frame time t_F in any given time step k using a multivariate function $t_{F,k}(f_k, \mathbf{x}_k(f_k))$. Besides showing the dependency of the frame time on the frequency and counters, this notation also reveals that the counters themselves can vary with frequency.

There are two major challenges in the characterization of $t_{F,k}(f_k, \mathbf{x}_k(f_k))$. The first challenge is to establish a trusted reference that provides a rich set of samples of this function. This set needs to provide the frame time for an exhaustive list of frequencies and counter values. The second and bigger challenge is to understand the sensitivity of frame time to frequency, *i.e.*, finding the partial derivative of the frame time with respect to the frequency. This information is vital for dynamic power management algorithms to find out how the performance would be affected by a change in the GPU frequency. However, finding the frequency sensitivity is very challenging, since it requires decoupling the impact of the change in frame time due to the frequency and frame complexity. In the rest of this section, we describe our solutions to address these challenges.

3.2 Frame Time and Counter Data Collection

Frame Time Measurement: Establishing the ground truth frame time is crucial for both developing the models and validating them later on benchmarks. Therefore, we modified Android's Direct Rendering Manager [6] driver to mark the times when the GPU starts and completes a new frame. This enables us to retrieve the frame time and frame count from the kernel at runtime.

Validation: To validate the correctness of our non-trivial modification, we also measured the platform power consumption using a data acquisition system. Figure 2a shows the total power consumed as a function of time when running a custom target application (Art3) at 60 frames per second (FPS). By maintaining a low CPU activity, we know that the peaks in the power consumption occur due to the GPU activity. For instance, the zoomed version of 50ms time in Figure 2b shows three frames as expected for 60 FPS and about 6ms frame time. Hence, we can test the accuracy of frame time and frame count instrumentations by correlating them with power measurements. Figure 2c shows the frame time probability density functions obtained by kernel instrumentation and power measurements. We observe that our kernel instrumentation and power measurements yield only 3% difference in mean frame time. We also find that the kernel instrumentation is more practical and accurate than the power measurements, since it does not depend on external equipment and suffer from measurement noise.

Data Collection: We used the Intel GPU tools [9] to log the counter values at runtime [11]. Our modified version of the kernel collects a trace in the format shown below:

Time	Frame Time	Frame Count	GPU Frequency	Perf. Cntr 1	Perf. Cntr 2	...	Perf. Cntr N

Each row corresponds to a 50ms interval, which matches the rate at which the frequency governors change the GPU frequency. We also tested that this data collection does not induce any noticeable impact on the application performance.

3.3 Decoupling the Impact of Frequency and Workload

One way to isolate the changes due to the GPU frequency is running the entire application repeatedly at each supported GPU frequency. Theoretically, the collected data could be used to identify the effect of GPU frequency on frame time. However, this approach is intractable for a number of reasons. First, there may not be a one-to-one correspondence between the frames in different runs. For example, consider an application that runs at 60 or 30 FPS depending on the GPU frequency. At the lower frame rate, the application will drop the 30 frames that it failed to render, rather than rendering them later. Second, even processing the same frame may take different amounts of time due to the variations in the memory access time from one run to another, as shown in Figure 3. We observe that frame time variations grow significantly even if the frame complexity changes marginally. These challenges are aggravated in many GPU intensive applications. Therefore, the most reliable approach to collect reference data is by varying the GPU frequency while freezing the workload, as described next.

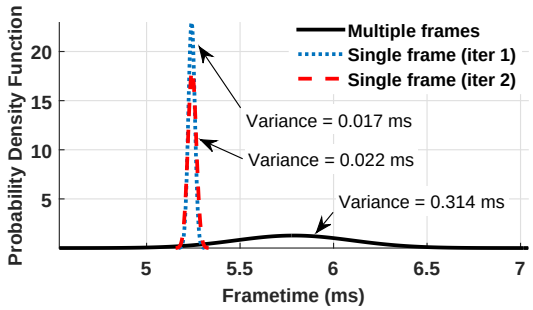

Figure 3: The frame time distribution obtained for rendering the same frame and rendering multiple similar frames.

3.4 Data Collection Methodology

As mentioned in the previous section, a consistent apple-to-apple comparison is possible only if the same frame is frozen and rendered repeatedly. To facilitate reference data collection, we built two custom Android applications, Art3 and RenderingTest, as detailed in Section 5.1. These applications enable us to precisely control the frame content and target frame rate.

The proposed data collection methodology is shown in Figure 4. We first set the CPU frequency for the repeatability of the results. Then, we sweep the GPU frequency across the set of frequencies supported by the target system. In our target platform, we used 9 frequencies ranging from 200MHz to 511MHz, as shown in Figure 4. Each of these combinations was further repeated for 64 frame complexities, which is determined by the number and variety of features

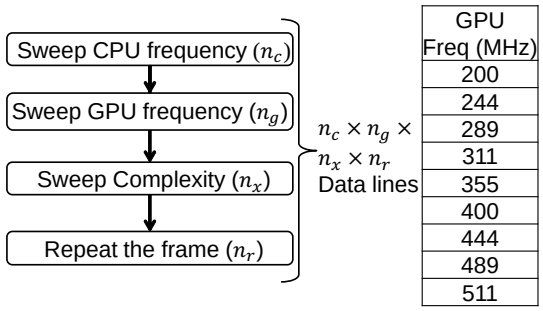

Figure 4: The proposed methodology for collecting a rich set of training and test data. Each frame is repeated n_r times for every configuration.

in a given frame. We note that different frame complexities enable us to exercise the performance counters in a controlled manner. Finally, we run each configuration multiple times to suppress the random variations. In our experiments, we collected 80 samples for each configuration, which led to $2 \times 9 \times 64 \times 80 = 92160$ lines with 1152 different configurations.

The proposed methodology is applied to both of our Art3 and RenderingTest applications. *Our data set confirms that the frame time is a function of both the GPU frequency and the workload.* For example, Figure 5 shows how the frame time changes with the GPU frequency at a CPU frequency of 1.3GHz. Different curves on this plot show that increasing frame complexity implies larger frame time, as expected. Similarly, Figure 6 shows the relation between the *Rendering Engine Busy* counter and the frame time. As the name implies, *Rendering Engine Busy* counts the number of cycles for which the rendering engine was active [11]. We observe that a larger cycle count (*i.e.,* higher complexity) results in an almost linear increase in frame time. Different curves on this plot also show that this counter itself is a function of the frequency, since it is counting the busy clock cycles.

In summary, our data set enables characterizing the multivariate function $t_{F,k}(f_k, \mathbf{x}_k(f_k))$. We use this data at design time to construct a template for the frame time model. Then, our online learning algorithm updates the coefficients in this model to predict the frame time for arbitrary applications.

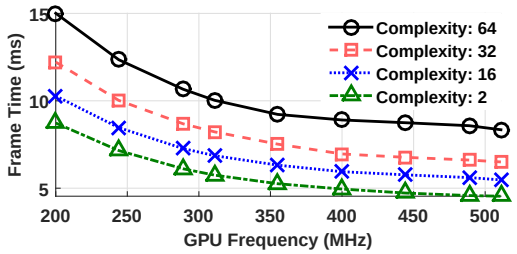

Figure 5: Frame time for the RenderingTest application with increasing GPU frequency at different frame complexities.

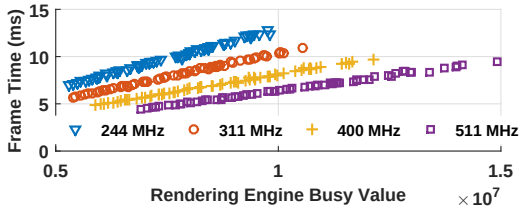

Figure 6: Frame time for the Rendering Test application with increasing complexity for four different GPU frequencies.

4. FRAME TIME PREDICTION

This section presents the proposed frame time prediction methodology. We first derive a mathematical model to express the change in frame time. Then, we describe the offline learning process needed to select the features that will be used during online learning. Finally, we present the proposed adaptive frame time prediction algorithm.

4.1 Differential Frame Time Model

The frame time at any given instant k can be obtained by summing up the measured frame time in the previous instant $k-1$ and the change in the frame time. This change can be approximated as a function of the GPU frequency and performance counters using partial derivatives as follows:

$$dt_F(f_k, \mathbf{x}_k(f_k)) = \frac{\partial t_F(f_k, \mathbf{x}_k(f_k))}{\partial f_k} df_k + \sum_{i=1}^{N} \frac{\partial t_F(f_k, \mathbf{x}_k(f_k))}{\partial x_i(f_k)} dx_{i,k}(f_k) \quad (1)$$

This equation reveals that the variation in frame time is a combined effect of the change in the GPU frequency (the first term), and the changes in the counters, which reflect the workload (the summation term). Equation 1 holds, if the frequency and counters are continuous variables. Since they are discrete variables in practice, we can approximate the change in frame time as:

$$\Delta t_F(f_k, \mathbf{x}_k(f_k)) \approx \frac{\partial t_{F,k}}{\partial f_k} \Delta f_k + \sum_{i=1}^{N} \frac{\partial t_{F,k}}{\partial x_{i,k}} \Delta x_{i,k} \quad (2)$$

Note that $\partial t_F / \partial f_k$ is the partial derivative of frame time with respect to frequency. The frame time change due to $\partial x_{i,k}(f_k)/\partial f_k$ is included in the difference term $\Delta x_{i,k}$. This equation forms the basis of our mathematical model. The differential form is useful, since the current frame time is known, and we are interested in the change. Moreover, it utilizes the difference of counters, which alleviates the need for feature normalization. Next, we analyze each term in detail to derive our frame time model.

Change due to the GPU frequency: In general, the part of the processing time confined within the GPU pipeline is inversely proportional with the frequency. However, memory access and stall times do not scale with the frequency. Therefore, the frame time is a nonlinear function of the GPU frequency, as shown in Figure 5. Using this observation, we can approximate the frame time t_F *for a given workload* (*i.e.* $\mathbf{x}$) in terms of a frequency scalable portion $t_{F,s}$ and an unscalable portion $t_{F,us}$ [2]. More specifically,

$$\begin{aligned} t_F(f_{k-1}, \mathbf{x}) &= t_{F,s}(f_{k-1}, \mathbf{x}) + t_{F,us}(\mathbf{x}) \\ t_F(f_k, \mathbf{x}) &= t_{F,s}(f_{k-1}, \mathbf{x}) \frac{f_{k-1}}{f_k} + t_{F,us}(\mathbf{x}) \end{aligned} \quad (3)$$

Hence, the change in frame time when jumping from f_{k-1} to f_k can be found by subtracting the first line in Equation 3 from the second line as follows:

$$\frac{\partial t_{F,k}}{\partial f_k} \Delta f_k \approx t_{F,s}(f_{k-1}, \mathbf{x}) \left(\frac{f_{k-1}}{f_k} - 1 \right) \equiv a_0 t_{F,k-1} \left(\frac{f_{k-1}}{f_k} - 1 \right) \quad (4)$$

where $t_{F,k-1}$ is the frame time from the previous instant $k-1$. We note that $t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right)$ can be easily calculated at run time. Since the scalable frame time is in general not known, we express it as *an unknown parameter* a_0 that our online learning algorithm will learn at runtime.

Hardware performance counter change: The frame time changes linearly with many hardware performance counters, such as the one shown in Figure 6. If any counters cause a non-linear change in frame time, they can be taken as piecewise linear. Thus, we express the second term in Equation 2, *i.e.,* the change in frame time with counters as:

$$\Delta t_F(\mathbf{x}_k) \approx \sum_{i=1}^{N} \frac{\partial t_{F,k}}{\partial x_{i,k}} \Delta x_{i,k} \equiv \sum_{i=1}^{N} a_i \Delta x_{i,k} \quad (5)$$

where a_i's are the coefficients that change at runtime as a function of the workload. Therefore, they are learned online.

By combining Equation 4 and Equation 5, we can re-write our mathematical model in Equation 2 as:

$$\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) \approx a_0 t_{F,k-1} \left(\frac{f_{k-1}}{f_k} - 1 \right) + \sum_{i=1}^{N} a_i \Delta x_{i,k}(f_k) \quad (6)$$

We use Equation 6 for online frame time prediction. The terms $t_{F,k-1}\left(\frac{f_{k-1}}{f_k} - 1\right)$ and $\Delta x_{i,k}(f_k)\ \forall\ i$ form the feature set $\mathbf{h}_k$, while the parameters $\mathbf{a} \in \mathbb{R}^{N+1}$ are learned online.

4.2 Feature Selection

Real-time prediction requires an extremely efficient learning algorithm to facilitate fast evaluation of a GPU frequency change. One approach to reduce the overhead of regression is dimensionality reduction on the input data. The goal of this approach is to reduce the complexity of the data and speed up computation, while maintaining a good prediction accuracy. In addition to algorithm efficiency, this can help remove the features that either add duplicate information to the output or do not change with our parameters. There are several reduction techniques including Least Absolute Shrinkage and Selection Operator regression (Lasso), Sequential Feature Selection (SFS), and Principle Component Analysis (PCA), which reduce the feature size in the model appropriately by selecting the most representative set of features. Choosing a specific technique for feature selection can depend on the data and application area.

Lasso regression: We used Lasso regression to minimize the mean squared error (MSE) with a bound on the l_1 norm of parameters a_i [7]. The results from Lasso regression are highly sparse due to l_1 nature of the bound. Therefore, if less sparsity is required, it is also possible to use elastic nets or ridge regression by varying the distribution of l_1 and l_2 norm penalties on the learning parameters. For P samples the Lasso regression can be performed by minimizing the MSE between the actual change in frame time $\Delta t_{F,k}$ and using the estimate from Equation 6 after adding a l_1 norm penalty as:

$$\hat{a} = \underset{a}{\operatorname{argmin}} \sum_{k=1}^{P} \left[\Delta t_{F,k} - a_0 t_{F,k-1} \left(\frac{f_{k-1}}{f_k} - 1 \right) - \sum_{i=1}^{N} a_i \Delta x_{i,k}(f_k) \right]^2 + \lambda \sum_{j=0}^{N} |a_j| \quad (7)$$

By increasing the value of λ, less features can be selected at the expense of accuracy. An acceptable loss in accuracy is about one standard error more than the minimum MSE.

Sequential Feature Selection (SFS): The SFS algorithm is a heuristic that adds features to an empty selection set in a stepwise manner to minimize the MSE in the prediction of a variable like GPU frame time. The result from SFS is close to the result of Lasso regression, but SFS is completely

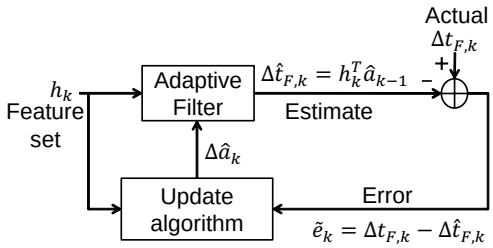

Figure 7: Adaptive filtering approach showing the update in parameters a_i based on error between the actual change in frame time and prediction.

oblivious to the multi-collinearity in the feature set. Hence, it is not an ideal methodology when the features are correlated. However, it can be faster to use in the case where the feature set is very large and known to have mostly uncorrelated features [20].

Principle Component Analysis: The PCA algorithm can help remove the low variance dimensions by centering, rotating and scaling the data along the eigenvectors. The eigenvectors corresponding to larger eigenvalues are retained and the rest are pruned. The retained eigenvectors are then used for transforming the original data [13]. One drawback of PCA is that it gives features in the transformed domain.

We implemented Lasso, SFS and PCA. In what follows, we present the results obtained with Lasso, because our goal is to achieve high sparsity on a correlated feature set while preserving the original meaning of the features. Thus, during the learning phase we will regress on M feature subset, where $M << N+1$, instead of $N+1$ features. Note that a trivial method to choose the number of features can lead to an increase in overhead or poor predictions.

4.3 Online Learning

The parameters in Equation 6 can be learned offline and then used at runtime. However, it is hard to generalize offline learning to all possible applications that would be executed by the system. Moreover, the workload can change as a function of user activity. Therefore, the learning mechanism should not completely rely on offline learning. We employ an adaptive algorithm to learn the parameters of the frame time model. In particular, we use the Recursive Least Squares estimation technique [15]. RLS algorithm updates the parameters a_i in Equation 6 in each prediction interval, as described in Figure 7, using the following set of equations:

$$\hat{a}_k = \hat{a}_{k-1} + \mathbf{G}_k[\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k)) - \mathbf{h}_k^T \hat{a}_{k-1}] \quad (8)$$

$$\mathbf{G}_k = \mathbf{P}_{k-1}\mathbf{h}_k[\mathbf{h}_k^T \mathbf{P}_{k-1}\mathbf{h}_k + 1]^{-1} \quad (9)$$

$$\mathbf{P}_k = [I - \mathbf{G}_k \mathbf{h}_k^T]\mathbf{P}_{k-1} \quad (10)$$

The update rule given in Equation 8 computes the prediction error by subtracting the frame time prediction from the actual change in frame time. Note that online learning would not be possible without our kernel instrumentation, which provides *reliable reference measurement at runtime* ($\Delta t_{F,k}(f_k, \mathbf{x}_k(f_k))$). Equation 9 and Equation 10 update the gain $\mathbf{G}_k$ and covariance $\mathbf{P}_k$ matrices using the feature vector. We refer the reader to [15] for details of the RLS algorithm.

Computational complexity: RLS is well known for giving good predictions in the signal processing field, however, its computational complexity grows with number of features as $O(M^2)$ [18]. Nonetheless, feature selection minimizes the size of the feature set to reduce the complexity. In particular, when the number of features shrinks from 34 to 4, the computational complexity reduces by about 70 times. Furthermore, matrix inversions are the main source of complexity in many algorithms, including RLS. Our solution is to use the co-variance form of RLS which does not perform matrix inversion. The value $\mathbf{h}_k^T \mathbf{P}_{k-1}\mathbf{h}_k$ in Equation 9 evaluates to a scalar, eliminating the overhead of the inversion operation.

4.4 Frame Time Sensitivity

Previous section explained how we predict the change in frame time $\Delta t_{F,k}(f_{k-1} \rightarrow f_k)$ by continuously learning the parameters $\widehat{\mathbf{a}}_{k-1}$ and our feature set. DPM algorithms often need to evaluate the impact of a frequency change on performance before making any decision. This information together with power sensitivity to frequency can help DPM algorithms to make better decisions. This section explains how our frame time prediction technique is used for this purpose.

As an example, consider a scenario where the GPU frequency at time k is $f_k = 400$ MHz. Suppose that a DPM algorithm needs to predict the change in frame time when the frequency goes from $f_k = 400$ MHz to a candidate frequency $f_{\text{new}} = 444$ MHz. Before finalizing this decision, the DPM algorithm needs to evaluate the corresponding change in frame time, *i.e.*, $\Delta t_{F,k}(f_k \rightarrow f_{\text{new}})$ using Equation 6. In this equation, the frequency change affects the first term $\left(\frac{400}{444} - 1\right)$ and only the counters that are a function of the frequency. To make the latter more explicit, we can write the change in counters due to the GPU frequency f and the frame complexity C as:

$$\Delta x_{i,k} \approx \frac{\partial x_{i,k}}{\partial f}\Delta f_k + \frac{\partial x_{i,k}}{\partial C}\Delta C, \ for \ \ 1 \leq i \leq N \quad (11)$$

Since the frame sensitivity is calculated for a given frame, the change in complexity $\Delta C = 0$, and Equation 6 can be written as:

$$\Delta t_F(f_k \rightarrow f_{\text{new}}) \approx a_0 t_{F,k-1}\left(\frac{f_k}{f_{\text{new}}} - 1\right) + \sum_{i=1}^{N} a_i \left(\frac{\partial x_{i,k}}{\partial f}(f_{\text{new}} - f_k)\right) \quad (12)$$

In Equation 12, f_k, f_{new}, and a_i are known at time step k. The only unknown value is $\frac{\partial x_{i,k}}{\partial f}$, which is zero for frequency *independent* counters. To model the derivative of the frequency *dependent* counters with respect to the GPU frequency, we can use a nonlinear function of frequency and frequency independent counters. Then, this model can also employ an online learning, such as RLS, or it can be learned offline. Subsequently, Equation 12 can be used to predict the change in frame time for the new candidate frequency as:

$$\frac{dt_F}{df}\Big|_k \approx \frac{\Delta t_F(f_k \rightarrow f_{\text{new}})}{f_{\text{new}} - f_k} \quad (13)$$

5. EXPERIMENTAL RESULTS

This section first describes the experimental setup and the results of offline feature selection. Then, we demonstrate the accuracy of the proposed online frame time prediction technique, and its potential impact on DPM algorithms.

5.1 Experimental Setup

We performed our experiments on the Minnowboard MAX platform [10] running Android 5.1 operating system with the

kernel modifications mentioned in Section 3.2. This platform has two CPU cores and one GPU, whose frequency can take the values listed in Figure 4. The GPU frequency is readily available from the kernel file system. In addition to this, we used the Intel GPU Tools as an external module to the Android system to trace the GPU performance counters.

Standard Benchmarks: We validated the proposed frame time prediction technique using the following commonly used GPU benchmarks: Nenamark2, BrainItOut, and 3DMark (both the Ice Storm and Slingshot scenarios).

Custom Benchmarks: The accuracy of the frame time prediction can be tested without any limitations, since our frame time prediction technique works for any Android app that can run on the target platform. However, validating the sensitivity prediction (*i.e.,* the derivative of the frame time with respect to the frequency) requires reference measurements taken at different frequencies. This golden reference cannot be simply collected by running the whole application at different frequencies due to the reasons detailed in Section 3.2. Therefore, we also developed RenderingTest and Art3 applications that enable us to control the number of times each frame is repeated.

The RenderingTest application accepts two inputs that specify the number of cubes rendered in the frame, and the number of times the same frame is processed. By changing the number of cubes, we control the frame complexity. In our experiments, we swept the number of cubes from 1 to 64, and repeated each frame 80 times. The cubes were rendered at a maximum of 60 FPS with vertex shaders and depth buffering enabled. Since we used the RenderingTest application for offline characterization, we developed one more custom application, called Art3, which renders pyramids with a different rendering pipeline. The RenderingTest application renders each cube with its own memory buffer, while Art3 concatenates all pyramids into the same memory buffer before rendering. The pyramids are not constrained by an FPS limit, but they are also rendered with vertex shaders and depth buffering. These two application allow us to compute and store the reference sensitivities, such that they can be used as the golden reference to validate our online frequency sensitivity predictions.

5.2 Feature Selection using Lasso Regression

We applied Lasso regression with 100–fold cross-validation on our large dataset collected from the RenderingTest application. Figure 8a shows the change in mean squared error between the predicted and measured frame time of the GPU. As λ in Equation 7 increases, the penalty on the cost function increases leading to higher MSE. The minimum value, $\lambda_{\text{min}} = 5.1 \times 10^{-4}$ uses all the features, as shown in Figure 8b. To shrink the model, a good choice is $\lambda_{\text{sel}} = 1.6 \times 10^{-1}$ for which the performance in terms of expected generalization error is about one standard error of the minimum. In our experiments, the number of features for λ_{sel} turns out to be four. The four selected features are change in the frequency term from Equation 6 and change in the *Aggregate Core Array Active*, *Slow Z Test Pixels Failing*, and *Rendering Engine Busy* counters. The Aggregate Core Array Active counter gives the sum of all cycles on all the GPU cores spent actively executing instructions. The Slow Z Test Pixels Failing counter gives the pixels that fail the slow check in the GPU. Neither of these counters depends on the frequency; they are functions of only the frame complexity. However, the Rendering Engine Busy counter changes with frame complexity, as well as frequency.

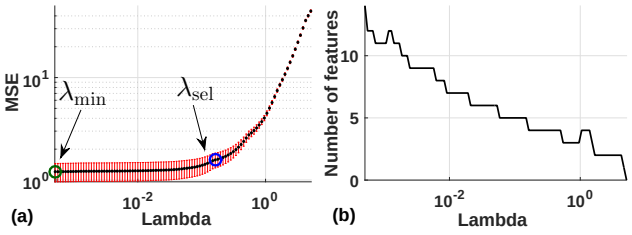

Figure 8: Cross-validated LASSO regression result for; (a) the change in mean squared error of the frame time prediction with increasing λ values, and (b) the change in the number of selected features with increasing λ values.

5.3 Online Frame Time Prediction

We validated our frame time prediction approach first on the RenderingTest application to test the corner cases. Figure 9 shows the comparison between the actual and the predicted frame time. During the first 5 seconds, both the GPU frequency and frames change randomly. We observe that the proposed online model successfully keeps up with the rapid changes. In order to test our approach under corner cases, we enforced a saw-tooth pattern during the remaining duration of the application. More precisely, the GPU frequency starts at 200 MHz, and the complexity increases from 1 to 64 in increments of one (the first tooth). Then, the same iterations are repeated for 9 supported GPU frequencies. Figure 9 demonstrates that we achieve very good accuracy when the frequency stays constant for a period of time. There is a spike when the complexity jumps suddenly from 64 to 1. However, the RLS reacts quickly and maintains a high accuracy. Overall, the mean absolute percentage error between the real and predicted frame time values is 2.1%.

We obtained similar levels of accuracy for Art3 and standard benchmarks. In particular, Figure 10 shows the actual and predicted frame times for 3DMark's Ice Storm benchmark at two different GPU frequencies. We achieved a high prediction accuracy with the mean absolute error of 2.8% and 7.9% for the GPU frequencies 200 MHz and 489 MHz, respectively. Similarly, the actual and predicted frame time for the BrainItOut gaming application with fixed GPU frequency is shown in Figure 11. This interactive game requires frequent user inputs, the frame time exhibits more sudden changes compared to other applications. Our frame time prediction matches closely to the actual frame time with the median and mean absolute percentage errors of 2.2% and 9.1%, respectively. Note that, the higher mean absolute error value for the BrainItOut application is due to a few outliers in the frame time. This is confirmed from the very low median absolute percentage error value of the benchmark.

The frame time prediction for all of the benchmarks run-

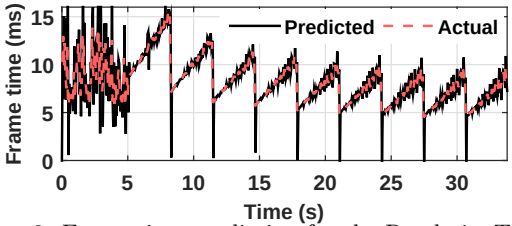

Figure 9: Frame time prediction for the RenderingTest app.

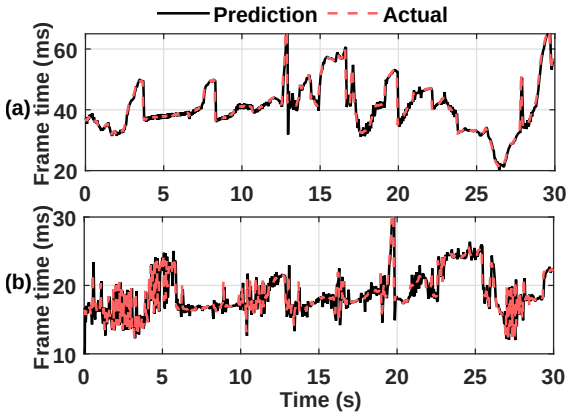

Figure 10: Frame time prediction for the 3DMark Ice Storm application running at (a) 200 MHz, (b) 489 MHz.

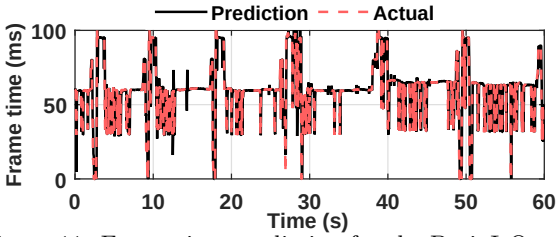

Figure 11: Frame time prediction for the BrainItOut application running at 200 MHz.

ning over *all GPU frequencies* is summarized in Figure 12. The average median and mean absolute errors across all the benchmarks are found as 1.3% and 3.8%.

We also compared our approach with an offline method, where all the model parameters are learned at design time and remained constant at runtime. Figure 13 shows the median absolute percentage errors for online (dashed line) and offline (solid line) learning for different training ratios. When we run all the benchmarks one after the other with our online learning mechanism, we get an error of 1.5%. However, running the same benchmarks with offline learned parameters leads to higher errors. As shown in the figure, the difference between the offline and online error decreases as the training ratio approaches one, *i.e.*, when the training set equals the test set. This shows that offline learning leads to higher error, unless the model can be trained on all the applications. Of note, the prediction error of our approach is flat, since the same set of features are selected with smaller training set.

5.4 Potential Impact for Dynamic Power Management

In this section, we demonstrate the accuracy of our frame time sensitivity prediction presented in Section 4.4. In our feature set for frame time prediction, only the *Rendering Engine Busy* counter is a function of frequency. After performing extensive analysis, we modeled the frequency dependence of this counter $\frac{\partial x_{\text{dep}}}{\partial f}$ empirically as a function of the frequency f, and two frequency independent counters *HIZ Fast Z Test Pixels Passing* and *3D Render Target Writes*.

$$\frac{\partial x_{\text{dep}}}{\partial f} \approx \sum_{i=1}^{2} \alpha_i x_{\text{indep}_i} + \beta_0 f + \beta_1 \sqrt{f} \qquad (14)$$

Then, we applied offline learning to characterize the α and

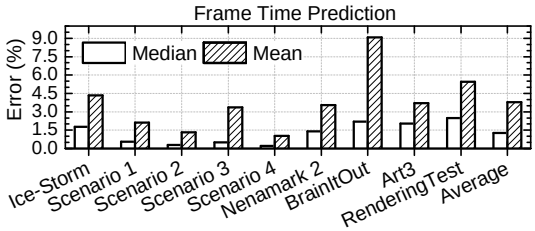

Figure 12: Median and mean absolute percentage errors in the frame time for the Android applications.

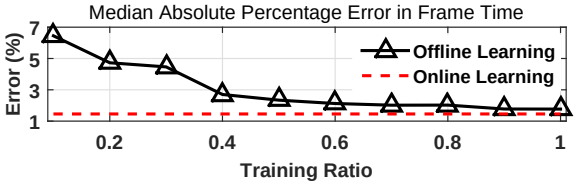

Figure 13: Comparison of median absolute percentage error in frame time for all Android applications combined.

β coefficients in this equation. We could use also online learning, but we opted for offline learning for three reasons. First, we observed that the change in the counters as a function of the frequency is much less dynamic than the frame time. Second, it is harder to obtain a clean reference for $\frac{\partial x_{\text{dep}}}{\partial f}$ at runtime, unlike the frame time which is obtained through instrumentation. Finally, this choice implies less computational overhead at runtime.

Figure 14 shows the real and predicted values of the derivative of this counter with respect to frequency for RenderingTest application. The root mean squared error of our prediction is 0.03, while the data range is $[-0.6, 0.4]$. Thus, Equation 14 provides a good approximation of this derivative.

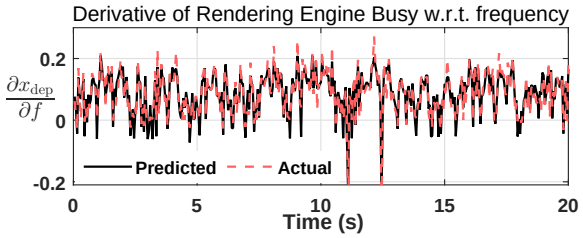

Figure 14: Offline prediction of the derivative of Rendering Engine Busy counter with respect to GPU frequency.

To assess the accuracy of our sensitivity prediction, we predict the change in frame time as a result of increasing (or decreasing) the frequency. Then, we compute the frame time sensitivity using Equation 13. We started with changing the frequency by one level according the supported GPU frequencies listed in Figure 4, *e.g.,* changing f_{GPU} from $f_k = 400$ MHz to $f_{new} = 444$ MHz or $f_{new} = 355$ MHz. Figure 15 shows the predicted and actual frame time when the new frequency f_{new} is one level higher. The mean absolute percentage error for this prediction is 1.5%. We observed similar results when f_{new} is one level lower. One might argue that the high prediction accuracy is only due to single frequency jumps like 400 MHz to 444 MHz. Therefore, we repeated our experiments for multiple frequency jumps. For example, if current frequency is 200 MHz, then a frequency jump of three implies f_{new} is 311 MHz. Figure 16 shows that the accuracy indeed degrades, but even when the number of frequency levels is six, the error is less than 7.5%.

We present the accuracy in predicting the derivative of

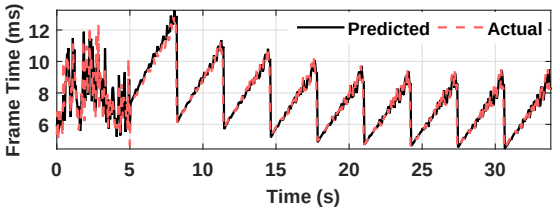

Figure 15: Predicted and actual frame times for RenderingTest application when f_{new} is one level higher.

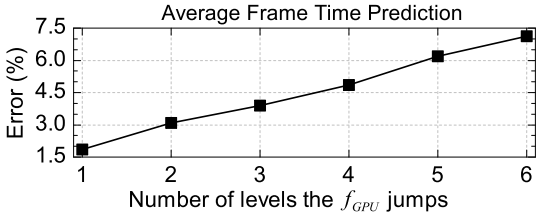

Figure 16: Frame time prediction error in Rendering Test application for multiple frequency jumps.

frame time with respect to GPU frequency for the RenderingTest application in Figure 17 . The root mean squared error in these predictions are 4.0×10^{-3} and 4.4×10^{-3} for frequency jumps of one level higher and lower, respectively. As seen from this plot, the slope starts with a negative value and then diminishes to zero on increasing frequency. This is consistent with the observation in Figure 6.

In addition to running the RenderingTest application we ran Art3 as well to measure frame time sensitivity. Figure 18 shows that the predicted derivative of frame time with respect to GPU frequency follows the reference values closely. In particular, the root mean squared error for the frame time sensitivity to frequency were 2.3×10^{-3} and 2.7×10^{-3} for frequency jumps of one level higher and lower, respectively.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a methodology that combines offline data collection and online learning. We constructed an RLS based adaptive runtime performance model using this methodology. Extensive evaluations on a commercial platform using common GPU benchmarks resulted in average mean absolute errors of 3.1% in frame time and 3.9% in frame time sensitivity prediction. This high accuracy model can help predict the sensitivity of the frame processing time to frequency, which is important for DPM algorithms. As future work, we plan to integrate the proposed runtime model into a sophisticated DPM algorithm.

Acknowledgments: This work was supported partially by Strategic CAD Labs, Intel Corporation and National Science Foundation under Grant No. CNS-1526562.

7. REFERENCES

[1] App Tornado. App Brain. http://www.appbrain.com/, accessed July 20, 2016.

[2] R. Z. Ayoub et al. OS-level Power Minimization under Tight Performance Constraints in General Purpose Systems. In *Proc. of the Intl. Symp. on Low-power Electronics and Design*, pages 321–326, 2011.

[3] W.-M. Chen, S.-W. Cheng, P.-C. Hsiu, and T.-W. Kuo. A User-Centric CPU-GPU Governing Framework for 3D Games on Mobile Devices. In *Proc. of ICCAD*, pages 224–231, 2015.

[4] B. Dietrich and S. Chakraborty. Lightweight Graphics Instrumentation for Game State-Specific Power Management in Android. *Multimedia Systems*, 20(5):563–578, 2014.

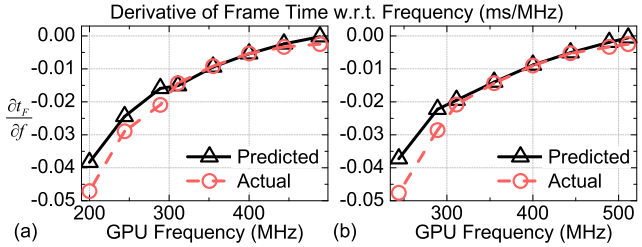

Figure 17: Sensitivity of frame time to frequency for RenderingTest app. with f_{new} one level; (a) higher, (b) lower.

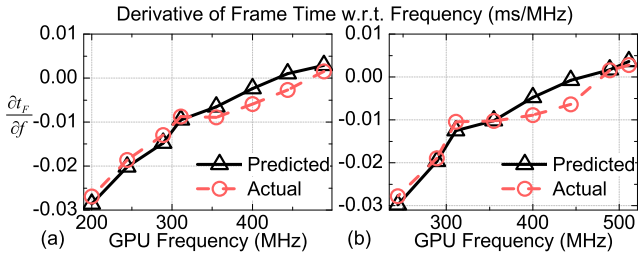

Figure 18: Sensitivity of frame time to frequency for Art3 application with f_{new} one level; (a) higher, (b) lower.

[5] B. Dietrich et al. LMS-based Low-complexity Game Workload Prediction for DVFS. In *Proc. of the Intl. Conf. on Comp. Design*, pages 417–424, 2010.

[6] R. Faith. The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure, 1999. http://dri.sourceforge.net/doc/drm_low_level.html, accessed July 20, 2016.

[7] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning*, volume 1. Springer Series in Statistics, Berlin, 2001.

[8] Future Mark. http://www.futuremark.com/benchmarks, accessedJuly20,2016.

[9] Intel Corp. Intel GPU Tools. http://01.org/linuxgraphics/gfx-docs/igt/, accessed July 20, 2016.

[10] Intel Corp. Minnowboard. http://www.minnowboard.org/, accessed July 20, 2016.

[11] Intel Corp. *Open Source HD Graphics Programmers' Reference Manual*. June 2015.

[12] T. Jin, S. He, and Y. Liu. Towards Accurate GPU Power Modeling for Smartphones. In *Proc. of the 2nd Workshop on Mobile Gaming*, pages 7–11, 2015.

[13] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[14] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz. A Control-Theoretic Approach for Energy Efficient CPU-GPU Subsystem in Mobile Platforms. In *Proc. of DAC*, pages 62:1–62:6, 2015.

[15] J. M. Mendel. *Lessons in Estimation Theory for Signal Processing, Communications, and Control*. Pearson Educ., 1995.

[16] H. Nagasaka et al. Statistical Power Modeling of GPU Kernels using Performance Counters. In *Proc. of the Intl. Green Computing Conf.*, pages 115–122, 2010.

[17] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra. Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs. In *Proc. of DAC*, pages 201:1–201:6, 2015.

[18] A. H. Sayed. *Fundamentals of Adaptive Filtering*. John Wiley & Sons, 2003.

[19] G. V. Varatkar and R. Marculescu. On-chip Traffic Modeling and Synthesis for MPEG-2 Video Applications. *IEEE Trans. on Very Large Scale Integration Systems*, 12(1):108–119, 2004.

[20] D. Ververidis and C. Kotropoulos. Sequential Forward Feature Selection with Low Computational Cost. In *Proc. of the European Signal Processing Conf.*, pages 1–4, 2005.