Managing Power for Closed-Source Android OS Games by Lightweight Graphics Instrumentation

Benedikt Dietrich, Samarjit Chakraborty
Institute for Real-time Computer Systems
Technical University of Munich, Germany
Email: {dietrich, samarjit}@rcs.ei.tum.de

Abstract—**Power consumption and battery life are important design concerns for mobile platforms. On these devices games can be considered as one of the most demanding applications in terms of computational cost and consumed energy. In this demo we showcase Android-based power management for games. We reduce the power consumption of games by scaling the processor's voltage and frequency. Towards this, the game's future workload has to be predicted. To accurately predict the workload, previous work heavily instrumented the game's source code itself. The source code is typically not available for up-to-date Android games. The work presented in this paper does not require any modification of the game's source code and therefore can as well be applied to closed source games. Towards this, we utilize the game's communication interfaces with the operating system to accurately predict a game's workload. The approach presented in the following has been implemented and tested on the PandaBoard ES [12] and Galaxy Nexus mobile phone with a number of popular closed-source games. Measurements show significant power savings while the gaming experience is maintained.**

I. Introduction

The increasing gap between the computational demand and battery capacity more than ever requires sophisticated power management algorithms. Especially, on portable devices like Android-based smartphones battery life is an important design concern. On these devices highly computational intensive games are a popular class of applications. Typically, games are not programmed in a power aware fashion, but are solely optimized for high frame rates and a good gaming experience. However, as shown in [3], frame rates above particular boundaries do not improve the gaming experience of most players and therefore are not required. In particular, for games played on relatively small displays like on smartphones the optimization for high frame rates can be questioned.

In our work we maintain a constant frame rate that is high enough to guarantee a good gaming experience. The frame rate is kept constant and power savings are achieved by scaling the processor's voltage and frequency up or down depending on the workload of future frames. Power management based on such prediction of the future workload is a very general approach applicable to both video and game applications. In case of games, accurate timing measurements of previously seen frames are required to predict the next frame's workload. To the best of our knowledge, all previous approaches are based on instrumenting the game's source code to gather the required timing information. The need for the source code's modification, however, restricts the applicability to a very small and not very meaningful selection of games.

In our demo we present an approach that does not require such modifications, but instead utilizes the game's communication interfaces with the underlying Android operating system. Thus, our method is applicable to any Android-based game. The demo is based on a PandaBoard ES [12] which has been modified to allow accurate power measurements. Additionally, a Galaxy Nexus phone is used to demonstrate the approach on a popular smartphone. Our setup opens up the possibility of evaluating any power management scheme on different closed-source Android-based games running on an ARM processor, which is representative of processors on portable devices like mobile phones.

II. Related Work

Previous work in the domain of application specific power management mainly focused on video playback applications. In this context, control theoretic approaches like PID controllers and more complex algorithms (see [1], [2], [6]) have been successfully utilized to predict the decoders upcoming workloads. These video-specific algorithms rely on buffering frames which is not a feasible approach for game applications.

Previous work on power management for games presented in [11] allowed the user to directly evaluate the current performance and the CPU has to be accordingly scaled up or down statically. Clearly, such approaches requiring user intervention will either lead to undesired frame rate drops in more demanding scenes or will not take full benefit of the potential power savings. Online workload prediction, similar to the video domain, has been shown in [5] and [4]. However, all of the above mentioned methods require the instrumentation of the game's source code. This in turn restricts the application range to a very small and not very meaningful choice of games. Moreover, it requires a good knowledge of the source code itself to be able to instrument the game at the right place. To the best of our knowledge, the approach presented in the following is the first approach to utilize the game's communication with the underlying OS and libraries for an accurate workload prediction and power management.

III. Design and Implementation

In the following section we will describe the implementation issues of our power management algorithm. First, we will give

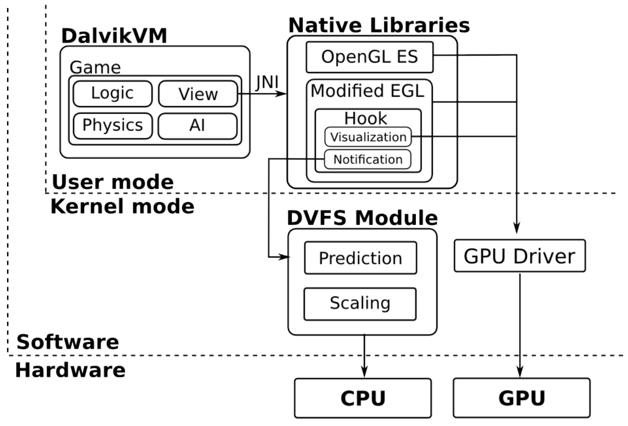

Fig. 1. System Architecture Overview.

a short introduction to the workload prediction technique for games that we use. Our setup of course can be used for other prediction techniques as well.

A. Time Series-based Workload Prediction

As described in [5], [4], the workload of a game's next frame is predicted based on the workload of previous frames. For example, using an Auto-regressive Model-based prediction, the next frame's workload is computed using the following equation:

$$\tilde{c}[i+1] = \sum_{k=0}^{n-1} w_k \; c[i-k], \tag{1}$$

where $c[i]$ is the measured execution time in cycles of the i-th frame and $\tilde{c}[i+1]$ the prediction for the next frame. The weights w_k are learned once and then kept constant. Thus, for the workload prediction the execution time of each frame needs to be measured. In the following we will describe how this execution time can be measured in Android without instrumenting the game's source code itself.

B. Android Implementation Details

Figure 1 gives an overview of the system architecture. Typically, Android games are written in Java and executed in their own Dalvik Virtual Machine. Besides computations like AI, physics and game logic, a game needs to render the game scene onto the screen. Dalvik provides the Java Native Interface (JNI) to allow games to make calls to native C/C++ libraries. The game uses this interface to call native OpenGL ES and Embedded Graphics Library (EGL) [10] functions. Calls are then forwarded to the GPU driver and finally to the GPU where the content is rendered to the so-called *back buffer*. Once the game has finished all computations for the current frame and issued all the required render calls, the `eglSwapBuffers()` function residing in the Embedded Graphics Library (EGL) is called by the game. This will cause the GPU to swap between the front and the back buffers. As a consequence the frame's content is shown on the display. It may be noted that in our work we assume that the game does not perform any frame rate control itself, but instead calls `eglSwapBuffers()` as often as possible in order to maximize the frame rate because a higher frame rate is directly associated with a better game experience for most games. This behavior has been observed for all the games used for evaluating our technique. In future, a game induced throttling could be detected by additionally taking the game's idle time into account by for example, detecting sleep-related calls.

In our work we leverage this standardized interface and instrument the `eglSwapBuffers()` function. Each time this function is called, we record a cycle accurate time stamp which then is used as input to our workload prediction. However, scaling the processor's frequency is only allowed in kernel mode. Towards this, we have implemented our own Android power management governor. At the load time this kernel module populates a character device to the system and creates a device node to allow user to kernel space communication. The first time a game issues an `eglSwapBuffers()`call, this device node is opened. In the following calls the opened node is used to send the recorded time stamps to the governor via `ioctl` syscalls. The governor receives the time stamps and performs the workload prediction according to Equation (1). Based on the prediction result and the desired target frame rate the required frequency is computed. The frequency is quantized to one of the available CPU frequencies and the scaling is initiated. An interface is provided to allow the configuration of the game's target frame rate. Moreover, for each known game a default target frame rate is stored which has been found to give a good gaming experience.

Our prediction algorithm is optimized for game applications. As not only game applications run on Android we detect the current application's type. This is done by comparing the name entry in `/proc/APPLICATION'sPID/status` with a provided list of known games. If the current application is found in the list, a game identification number is sent to the kernel module. Otherwise, the governor is notified that currently not a game has the focus. Depending on the application's type the governor will either perform game optimized power management or behave as default `Interactive` Android power management governor.

IV. Demo Setup and Proof of Concept

We have implemented the approach described above on two Android devices, namely the Pandaboard ES [12] attached to a 10" Multitouch LCD Display and a Samsung Galaxy Nexus. Both devices are based on an OMAP4460, a dual core ARM Cortex-A9 1.2GHz Mobile processor from Texas Instruments [9].

The demo includes the measurement setup depicted in Figure 2. We have modified the PandaBoard to allow power measurements with the help of shunt resistors. The Texas Instruments INA199 [8] amplify the corresponding voltage drops at the shunt resistors. The amplified voltage is then measured with the help of the analog digital converters of a Texas Instruments MSP430 [7] at a sampling rate of 50 kHz. A GPIO pin of the PandaBoard triggers the MSP430 to start the measurements synchronized with the start up of the game. Once the player exits the game, the MSP430 is again signaled

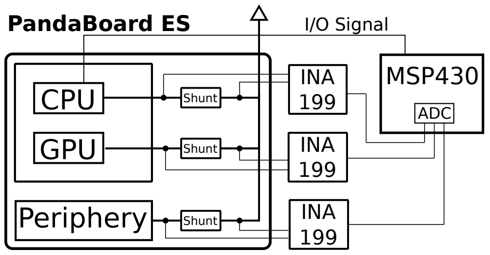

Fig. 2. Measurement Setup

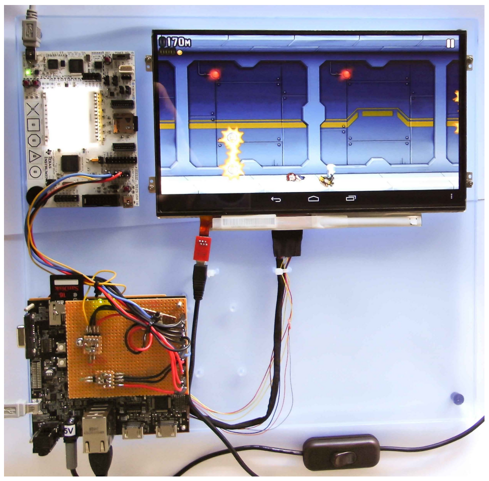

Fig. 3. Experimental Setup consisting of the modified PandaBoard (lower left), the Multitouch LCD (upper right) and the Power Measurement Unit (upper left).

TABLE I
OVERHEAD MEASUREMENT RESULTS

Type	Average Overhead
Read Cycle Counter	71 cycles
`ioctl` syscall	989.7 cycles
Workload Prediction	207.3 cycles
Voltage Frequency Scaling	241 - 852μs

and the measurement is stopped. The power measurement results are shown on the display attached to the MSP430. The real setup consisting of the modified PandaBoard, the Multitouch LCD and the power measurement unit is shown in Figure 3. The Android governor can be changed to allow a comparison between the default `Interactive` and our `Gaming` governor. In addition, we have used the Samsung Galaxy Nexus to show the implementation running on a commercial Android device. The setup has been tested with popular games like Dragonfly, Jetpack and TurboFly 3D on the PandaBoard and additionally on the Samsung Galaxy Nexus with the highly demanding Shadowgun game. Significant power savings could be observed without an visible impact on the gaming experience.

Overhead: the proposed implementation comes with a computational overhead. The results of the overhead measurements are given in Table I. Reading the ARM's cycle accurate time stamp takes 71 cycles in average. The `ioctl` command to the Android governor in average consumes 989.7 cycles. The workload prediction itself is performed in average within 207.3 cycles. Scaling of the voltage and frequency is the largest contributor to the total overhead and depends on the current and the target frequency. Measurements have shown that the switching time ranges from $241\mu s$ to $852\mu s$ per scaling. Assuming a game running with 30 frames per second, the total overhead in the worst case is 2.59 %.

V. CONCLUSION AND FUTURE WORK

In our work we have shown how Android can be extended to allow application specific power management for games. Previous work on power management for games has been restricted to open-source games as code instrumentations were required. In contrast, our proposed setup gives the possibility to evaluate power management algorithms for any Android-based game for which the source code is not available and hence cannot be instrumented. Further, we have modified the PandaBoard and integrated a low-cost power measurement device. As measurements are directly controlled by the Android OS, repeatability and high accuracy is guaranteed. The setup can now be used, to evaluate the efficiency of game specific power management governors for any Android-based closed source games running on an ARM processor.

Future plans target to instrument other OpenGL calls and evaluate if OpenGL call patterns can be leveraged to further improve the workload prediction.

ACKNOWLEDGMENT

The work reported in this paper was partially funded through a joint project with the Microprocessor Lab of Intel Labs Germany. We specially thank Matthias Gries for his helpful comments.

REFERENCES

[1] A Acquaviva, L Benini, and B Ricco. An adaptive algorithm for low-power streaming multimedia processing. In *Design, Automation and Test in Europe (DATE)*, March 2001.
[2] K Choi, K Dantu, W.-C. Cheng, and M Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *International Conference on Computer-Aided Design (ICCAD)*, November 2002.
[3] M Claypool, K Claypool, and F Dama. The effects of frame rate and resolution on users playing First Person Shooter games. In *ACM/SPIE Multimedia Computing and Networking (MMCN)*, January 2006.
[4] B Dietrich, S Nunna, D Goswami, S Chakraborty, and M Gries. LMS-based low-complexity game workload prediction for DVFS. In *ICCD*, pages 417–424, 2010.
[5] Y Gu, S Chakraborty, and W T Ooi. Games are up for DVFS. In *Design Automation Conference (DAC)*, July 2006.
[6] C J Hughes and S V Adve. A formal approach to frequent energy adaptations for multimedia applications. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2004.
[7] Texas Instruments. MSP430F551x MSP430F552x Mixed Signal Controller. 2009.
[8] Texas Instruments. INA199A1 Voltage Output , High or Low Side Measurement , Bi-Directional Zero-Drift Series INA199A1. Technical report, 2010.
[9] Texas Instruments. OMAP4460 Data Manual. Technical Report January, 2012.
[10] Khronos Group. Embedded Graphics Library.
[11] A Mallik, B Lin, G Memik, P Dinda, and R P Dick. User-driven frequency scaling. *IEEE Computer Architecture Letters*, 5(2):16, July 2006.
[12] Pandaboard.org. Pandaboard ES - System Reference Manual. 2011.