TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Realzeit-Computersysteme

Power Management for Closed-Source Interactive Games on Mobile Devices

Benedikt Dietrich

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. habil. Gerhard Rigoll
Prüfer der Dissertation:	1. Univ.-Prof. Dr. sc. (ETH Zürich) Samarjit Chakraborty
	2. Prof. Preeti Ranjan Panda, Ph.D. Indian Institute of Technology Delhi, Indien

Die Dissertation wurde am 25.09.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 07.09.2015 angenommen.

Abstract

On mobile devices like smartphones and tablet PCs, games account for the class of most popular, but at the same time most power hungry applications. The processor of such devices is one of the main contributors to the total power consumption when games are being played. A common technique to reduce the processor's power consumption is dynamic voltage and frequency scaling. Power managers of modern operating systems reduce the processor's power consumption by scaling the processing frequency and voltage merely based on the processor's current utilization and without considering the requirements of running applications. Since high frame rates are commonly attributed to a good gaming experience, most games process as many frames as possible resulting in a high system utilization. Traditional power managers in turn select constantly high processing frequencies resulting in a high power consumption. Studies have shown that frame rates above a particular point don't improve the gaming experience. Hence today's gaming platforms perform not required computations and thereby waste a significant amount of energy.

In this thesis a power manager is presented that is aware of the game application, predicts its future computing requirements and efficiently selects the processor's clock frequency that guarantees the desired frame rate and at the same time provides optimal power savings. Core of this power manager is the workload prediction which is based on statistical relationships between processing times of past frames and the current frame. While predictors like the PID controller-based predictor have been widely studied for video decoding applications in the past, there is only little work on workload prediction for games. We systematically evaluate workload predictors and identify that techniques such as PID controller-based workload predictors lack the robustness to accurately predict the highly dynamic game workloads and result in poor power savings and a deteriorated user experience. Our results show that autoregressive model-based predictors – the most general form of linear models – provide significant power savings for all considered games (up to 32.4 % compared to Android's default power manager) while maintaining the quality perceived by the user.
While many multimedia applications like video decoders are open-source and can be directly instrumented to measure frame timings, which are required as input to the workload prediction, most modern games are closed-source. In this work we for the first time present a technique that does not only allow measuring the frame timings of any closed-source game, but as well can be used to identify a game's current state, e.g., if the game is viewing a menu, the loading screen or is in the actual gaming state. This state information is leveraged to implement a state-specific power management that exploits state-specific workload characteristics and requirements and thereby further reduces the power consumption by up to 43.2 % compared to Android's default power manager.

The discussed techniques have been successfully verified for modern closed-source games from different genres. We have developed measurement setups that allow detailed measurements of system parameters and the power consumption for three different operating systems (Linux, Windows and Android). For all platforms highly portable and widely applicable solutions have been found and significant power savings were obtained. Based on a user study we verified that developed methods don't deteriorate the gaming quality experienced by the user. By investigating the theoretical limits of power savings and modeling an optimal workload predictor, we reveal the remaining gap of our approach to the theoretical optimal power manager and thereby motivate future research endeavors.

In summary, the main contributions of this thesis are: i) We identified a suitable workload prediction technique for highly variable game workloads which guarantees significant power savings while maintaining the gaming quality. ii) We present a technique which for the first time allows not only applying the developed power management scheme to closed-source games, but as well identifying the game's current state. iii) The developed measurement setups provide detailed insight into the current state of the systems and their power consumption. Based on these setups, we verified the described algorithms and showed significant power savings of up to 43.2 % compared to Android's default power manager. iv) We determined the theoretical limits of power savings, revealed the gap between our approach and the theoretical optimum and thereby motivate future research in the domain of game power management.

Kurzfassung

Spiele zählen auf mobilen Endgeräten wie Smartphones und Tablet PCs zu den beliebtesten Anwendungen, weisen allerdings auch den größten Energiebedarf auf. Im Falle von Spielanwendungen wird der größte Teil der Energie vom Prozessor verbraucht. Power Manager heutiger Betriebssysteme passen die Rechenfrequenz und Versorgungsspannung dynamisch an die aktuellen Rechenanforderungen an, um den Energieverbrauch des Prozessors zu reduzieren. In der Regel wählt der Power Manager hierzu die benötigte Rechenfrequenz allein basierend auf der momentanen Auslastung des Prozessors aus, ohne die realen Anforderungen laufender Anwendungen zu berücksichtigen. Üblicherweise berechnen Spiele so viele Bilder pro Sekunde wie möglich, da eine hohe Bildwiederholrate generell mit einem guten Spielerlebnis assoziiert wird. Dies resultiert in einer hohen Systemauslastung, auf Grund welcher herkömmliche Power Manager durchgehend hohe Rechenfrequenzen verwenden, die wiederum in einem hohen Energieverbrauch resultieren. Studien haben hingegen gezeigt, dass sehr hohe Bildwiederholraten das Spielerlebnis nicht verbessern. Somit werden auf heutigen Spieleplattformen unnötig viele Bilder berechnet und Energie verschwendet.

Im Rahmen dieser Arbeit wurde ein Power Manager entwickelt, der sich der laufenden Spielanwendung bewusst ist, die zukünftig benötigte Rechenzeit der Anwendung vorhersagt und die Rechenfrequenz so skaliert, dass eine gewünschte Bildwiederholrate auf energieeffiziente Art und Weise garantiert wird. Herz des Power Managers ist die Vorhersage künftiger Rechenzeitanforderungen, welche auf statistischen Abhängigkeiten zwischen aktuellen und vergangenen Rechenzeiten basiert. Während Prädiktoren, wie PID Controller-basierte Prädiktoren, in der Vergangenheit umfassend für die Rechenzeit-Vorhersage von Videodekodern untersucht wurden, gibt es nur wenige Arbeiten, die sich mit der Vorhersage von Spieleanforderungen beschäftigen. Diese Arbeit evaluiert Rechenzeit-Prädiktoren systematisch und zeigt auf, dass Techniken, wie etwa der PID Controller-basierte Prädiktor, nicht für die Vorhersage von stark variierenden Rechenzeiten von Spielen geeignet sind. Unsere Analyse zeigt weiterhin, dass Prädiktoren, die auf autoregressiven Modellen basieren – die allgemeinste Form linearer Modelle – für alle getesteten Spiele erhebliche Energieersparnisse von bis zu 32.4 % im Vergleich zu Android's eigenem Power Manager erzielen, ohne die Spielqualität zu beeinträchtigen.
Während der Quelltext vieler Multimediaanwendungen, wie etwa Videodekodern, frei erhältlich ist und direkt instrumentiert werden kann, um die Rechenzeit einzelner Bilder zu messen, ist der Quelltext der meisten aktuellen Spieletitel nicht erhältlich. In dieser Arbeit wird eine Methode präsentiert, die es nicht nur erlaubt, diese Rechenzeiten für beliebige Closed-Source Spiele zu messen, sondern auch die Möglichkeit bietet, den aktuellen Zustand des Spiels zu ermitteln. Es wird zum Beispiel erkannt, ob das Spiel momentan

einen Ladebildschirm, ein Menü oder den tatsächlichen Spielinhalt anzeigt. Dieses Wissen wird verwendet, um für jeden Zustand eine spezifische Power Management Strategie anzuwenden und somit den Energiebedarf weiter zu senken (um bis zu 43.2 % im Vergleich zu Android's eigenem Power Manager).
Beschriebene Methoden wurden erfolgreich für aktuelle Closed-Source Spiele verschiedener Genres getestet. Im Zuge dessen wurden für drei verschiedene Betriebssysteme (Linux, Windows und Android) Messplattformen entwickelt, die das detaillierte Aufzeichnen von Systemeigenschaften und des Energieverbrauchs ermöglichen. Für alle Plattformen wurden leicht portierbare, flexible Lösungen entwickelt und erhebliche Energieersparnisse erzielt. Basierend auf einer Nutzerstudie wurde verifiziert, dass der entwickelte Power Manager keinen Einfluss auf die Spielqualität hat. Zusätzlich wurden theoretische Grenzen analysiert, um das noch bestehende Potential an möglichen Energieersparnissen zu ermitteln und somit weiterführende Forschungsarbeiten in diesem Bereich zu motivieren.

Die wesentlichen Beiträge dieser Disseratation können wie folgt zusammengefasst werden: i) Es wurden Prädiktoren identifiziert, die für die Vorhersage von stark variierenden Rechenzeitanforderungen geeignet sind. Diese garantieren in Verbindung mit dem entwickelten Power Manager erhebliche Energieersparnisse ohne das Spielerlebnis zu beeinträchtigen. ii) Es wurde zum ersten mal eine Ansatz aufgezeigt, der es nicht nur erlaubt, entwickelte Power Management Methoden für Closed-Source Spiele anzuwenden, sondern es auch ermöglicht, den aktuellen Zustand des Spiels zu detektieren und dieses Wissen auszunutzen, um den Energiebedarf weiter zu senken. iii) Die entwickelten Messplattformen erlauben einen detaillierten Einblick in den aktuellen Zustand des Systems und dessen Energieverbrauch. Mittels dieser Plattformen wurden die beschriebenen Algorithmen verifiziert und erhebliche Energieersparnisse von bis zu 43.2 % im Vergleich zu klassischen Ansätzen aufgezeigt. iv) Es wurde das verbleibende Potential an theoretisch bestmöglichen Energieersparnissen ermittelt und hierdurch weiterführende Forschung motiviert.

Acknowledgements

I would like to thank several people for their support and contributions to this work. First of all, I would like to express my very great appreciation to my advisor Samarjit Chakraborty for his support and guidance during the past five years. I consider myself as very fortunate for having such an excellent and insightful advisor. I would also like to thank Preeti Ranjan Panda for being my second reviewer. Although we never collaborated, we had very productive and fruitful discussions on game power management and related topics.

This work has been partially funded through a joint project with Intel Labs Braunschweig and I wish to acknowledge the help provided by Mathias Gries who always gave valuable and extensive feedback on my work. I am as well very grateful for the very productive collaboration with Apratim Guha and the students I had the pleasure to work with on game power management: Runhua Xu, Stefan Theil, Rohit Bisani and Stasys Hiob.

During my time as doctoral student, I had the great opportunity to work at NVIDIA in Santa Clara, California. I would like to thank Santanu Dutta and Samarjit Chakraborty for providing this great experience. During this research co-op, I acquired practical skills and deep insight into graphics driver development which proved to be very beneficial for my research and future career.

I am very grateful for having had such great colleagues during the past five years, who made the time more than enjoyable. All the good laughs and fun discussions we had in the Mensa made even the Mensa food look appealing and the coffee in the kitchen enjoyable. A special acknowledgement goes to Reinhard Schneider, Daniel Yunge, Michael Balszun and my former office room-mate Martin Schäfer. You became true friends for me and I am hoping to still regularly catch up with you for climbing, swimming and/or a relaxed beer or two. Also thank you Martin Geier, Reinald Gfüllner and Martin Becker for the great IT support. Special thanks goes to Florian Rattei and Robert Diemer and the administrative staff of our institute. You have been extremely helpful and supportive over the last years.

I particularly thank my friends, my parents, my sister Laura and my brother Simon for their love, support and continuing encouragement.

Munich, August, 2014

Contents

List of Symbols

ACPI	Advanced Configuration and Power Interface
AI	Artificial Intelligence
AMOLED	Active Matrix Organic Light Emitting Diode
API	Application Programming Interface
AR	Autoregressive
ARMA	Autoregressive Moving Average
CMOS	Complementary Metal-Oxide-Semiconductor
CPI	Cycles per Instruction
CPU	Central Processing Unit
DPM	Dynamic Power Management
DTM	Dynamic Thermal Management
DVFS	Dynamic Voltage and Frequency Scaling
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communications
HVS	Human Visual System
I/O	Input/Output
IC	Instruction Count
IPC	Instructions per Cycle
LCD	Liquid Crystal Display
LKM	Linux Kernel Module
LMS	Least Mean Squares
MAR	Memory Access Rate
MC	Motion Compensation
MOS	Metal-Oxide-Semiconductor
MVS	Multi-Level Voltage Scaling
NLMS	Normalized Least Mean Squares
OLED	Organic Light Emitting Diode
PCM	Phase Change Memory
PID	Proportional-Integral-Derivative
RAM	Random Access Memory
SVS	Static Voltage Scaling
VLD	Variable Length Decoding
WNIC	Wireless Network Interface Card

1

Introduction

The popularity of powerful mobile devices like smartphones and tablet PCs has significantly increased over the last years. In 2013 alone, 1 billion smartphones have been sold worldwide [45]. On such devices, power consumption is a major design concern and heavily influences the purchasing decision of customers. According to a study performed by Qualcomm [1], the key factors price and promotion are closely followed by battery life when customers are asked about their buying decision. In the German study by the Institut für Demoskopie Allensbach in 2013 [2], 75.8 % of interviewed persons named battery life as one of the key criteria when buying a new smartphone.

On such devices, compute-intensive games are one of the most popular class of applications. Among all categories of applications available in the iTunes store, games have the largest share with 16.49 % [3]. On an average, 137 new games have been submitted per day to the iTunes store in October 2012. Games account for 67 % of time spent using tablets and 39 % for smartphones [47].

At the same time, graphics intensive games are highly demanding applications in terms of processing time resulting in a significant amount of power being consumed by the processor. Operating systems like Android or Windows implement techniques to reduce the processor's power consumption. These techniques, however, are completely unaware of the running applications and their requirements. Figure 1.1 shows the normalized power consumption using Android's default power manager and the theoretically optimal power manager. The optimal power manager provides the theoretical minimum possible

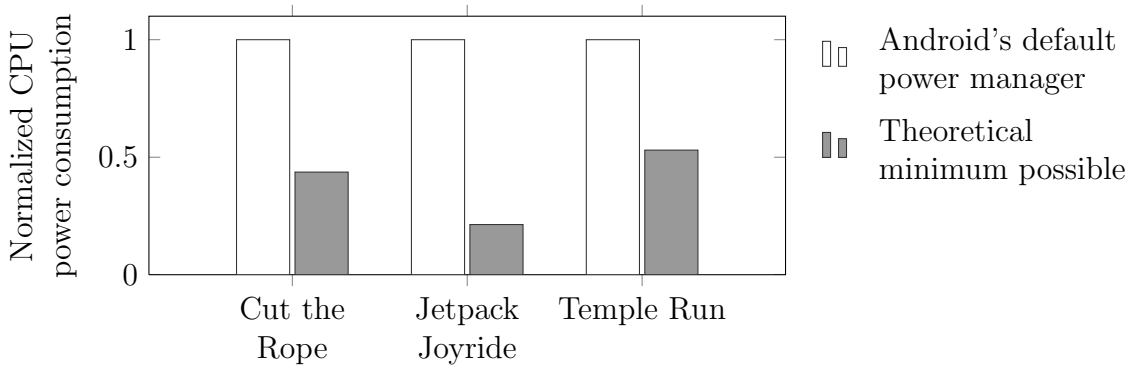


Figure 1.1: Normalized power consumption using Android's default and the theoretical minimum possible power consumption

power consumption while maintaining the best possible user experience. Clearly, there is a huge gap between currently employed power managers and the theoretically achievable optimum. This work targets to close the existing gap and thereby prolong the battery life of mobile devices.

The rest of this chapter is organized as follows. We first describe different components of a processor's energy consumption and a popular technique that allows reducing the energy consumption, namely dynamic voltage and frequency scaling (DVFS). While DVFS is widely used and an integral part of the Linux Kernel, current techniques are not suited to efficiently reduce the power consumption of the processor when games are being played. We explain the basic structure of games, point out the weaknesses of currently employed power managers to then describe the basic idea of game power management which this work is based on. Next, we detail challenges of our approach, the contributions of this work and the organization of the remainder of this thesis.

1.1 Dynamic voltage and frequency scaling

A CMOS circuit's power consumption is given by the dynamic and static (or leakage) power:

$$P = P_{dynamic} + P_{static} \approx CfV_{dd}^2 + P_{short-circuit} + V_{dd}I_{leak}, \tag{1.1}$$

where C is the load capacitance, f is the operating frequency, V_{dd} is the supply voltage, $P_{short-circuit}$ is the short-circuit power and I_{leak} is the leakage current.

The **static** power P_{static} is independent of the transistors' switching activities and is always consumed when the circuit is powered. While static power consumption could be neglected in the past, the leakage current significantly increased with the continuing technology scaling. One popular technique to reduce the static power consumption is *power gating*. Here, currently not required parts of the circuit are cut off from the supply voltage.

The **dynamic** power $P_{dynamic}$ is required to charge up the load capacitance and thereby change the logic state of CMOS transistors. In addition, state changes consume short-circuit power $P_{short-circuit}$ which is caused by a direct current flow from supply to ground due to NMOS and PMOS transistors being active at the same time when switching. The short-circuit power only contributes a small amount to the total power consumption and can be neglected [151]. With *clock gating*, a significant amount of dynamic power can be saved. The clock supply to unused logic parts of the circuit is simply disconnected. Alternatively, the clock frequency of the circuit can be lowered to reduce the dynamic power consumption. Slowing down, e.g., a processor, however, prolongs the processing time and hence might, due to leakage power, even increase the total amount of energy consumed to finish a particular task. Since a lower frequency results in larger circuit delays, the supply voltage can be lowered, too ($V_{dd} \propto f$), resulting in an approximately cubic reduction of the dynamic power. Hence, *dynamic voltage and frequency scaling* (DVFS) is a very attractive choice to reduce a processor's power consumption.

Most modern mobile processors, as well as desktop and server architectures, support all of the above described features, viz., DVFS, clock gating and power gating, to lower the power consumption. Interfaces are provided to control the hardware features in software. The Advanced Configuration and Power Interface (ACPI) specification standardized these interfaces [59] providing an OS-independent power management and configuration interface. For dynamic voltage and frequency scaling (DVFS) so-called *P-states* have been defined, i.e., valid voltage-frequency settings (operating points), where P0 defines the operating point at which the processor is running at the highest frequency and voltage. The voltage and frequency, as well as the power consumption decreases with an increasing P-state number. Besides the P-states, most modern hardware provides several C-states, i.e., CPU *idle states*. While the CPU is fully operational in state C0, deeper C-states (higher C-state numbers) halt and switch off parts of the CPU (using clock- and power-gating). Deeper C-states consume less power, but have a larger target residency, i.e., the time the system should at least spend in the corresponding state to save energy. Further, waking up from deeper C-states takes longer and hence impacts the response time of the system.

The above described power management techniques are an integral part of the Linux Kernel since version 2.6.0 (released in December 2003). The `cpufreq` and `cpuidle` Kernel subsystems are responsible for the P-state selection and potential C-state transitions. The so-called *governors* decide about future transitions based on system metrics like the processor utilization and forward the requests to hardware-specific drivers which then perform the actual voltage and frequency scaling and C-state transition. Since different types of processors and workloads might benefit from different scaling and idling strategies, Linux allows choosing among different governors.

The widely used `ondemand` governor [110] has been introduced in October 2004 with Kernel version 2.6.9. The original version shown in Algorithm 1 profiles the system utilization on a configurable millisecond basis. Once the utilization is larger than UP_THRESHOLD, the processor's frequency is increased to the maximum. If the system utilization falls below DOWN_THRESHOLD, the frequency is steadily decreased by 20 % until the utilization is again larger than the DOWN_THRESHOLD. This approach was later changed to jump

Algorithm 1 Original Linux `ondemand` governor

```
1: for all CPU in the system do
2:     every X milliseconds do
3:         Get utilization since last check
4:         if Utilization > UP_THRESHOLD then
5:             Increase frequency to MAX
6:         end if
7:         if Utilization < DOWN_THRESHOLD then
8:             Decrease frequency by 20 %
9:         end if
10:    end every
11: end for
```

directly to the lowest frequency that keeps the CPU at 80 % utilization instead of reducing the frequency steadily. Further, for multi-core processors that do not support the individual scaling of each core's frequency, the `ondemand` governor was changed to consider only the core with the highest utilization for the frequency decision. On Android-based mobile platforms the `interactive` governor is commonly used, which was designed for latency-sensitive, interactive workloads. Similar to `ondemand`, this governor selects the operating frequency based on the system's current utilization, but more aggressively scales to higher frequencies.

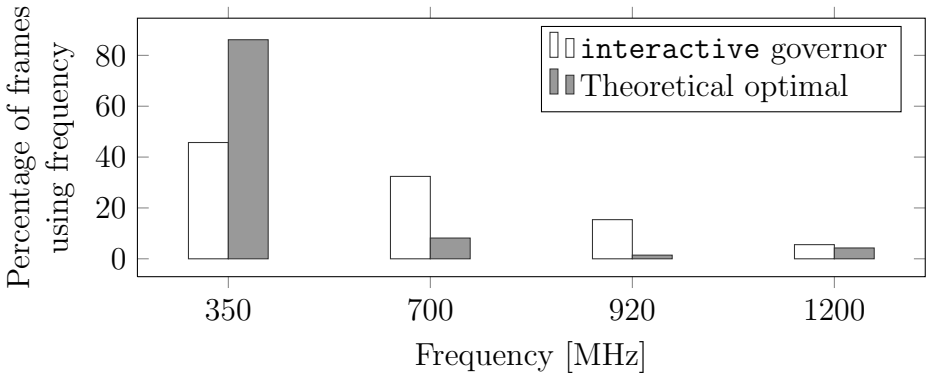


Figure 1.2: CPU processing frequencies that are used during a game play of Jetpack Joyride to maintain 58 frames per second

The described Linux governor significantly reduces the power consumption for most applications compared to running the processor always at the highest frequency. In the case of gaming applications, however, the utilization-based frequency selection turns out to be highly inefficient. As shown in Figure 1.2, the Android's `interactive` governor in many cases chooses frequencies that are too high compared to the theoretical optimal power manager. Reasons for this inefficiency, the general structure of games, and the solution proposed by this work are detailed in the following.

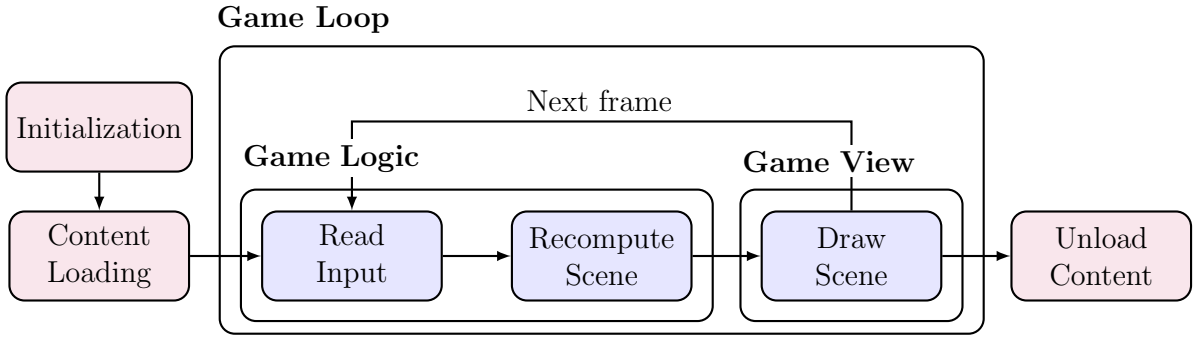


Figure 1.3: Typical execution flow of a game

1.2 General game architecture

A game typically consists of three main components [101]:

- The **game logic** forms the core of each game. It defines the game world, its objects and how they interact with each other. It takes external stimuli such as user input and elapsed time to recompute object positions in the virtual world, updates the artificial intelligence (AI), performs collision detection and simulates physics and particles.
- The **game view** communicates with the game logic to translate the game state into a rendered scene. Towards this, it typically issues commands to the graphics processing unit (GPU) which eventually renders the scene to the so-called back-buffer, i.e., a region in the memory that stores the final color values of a frame. Once the scene is composed, the current back-buffer becomes front-buffer (this is commonly referred to as buffer swap) and at the next display refresh, the memory content is read and shown on the display.
- Since most games are developed not only for one device, but should run on versatile hardware platforms and operating systems, the **application layer** abstracts hard- and software-specific implementation details to the upper layers such as the game logic and game view. Ideally, when porting a game from one platform to another, only the application layer has to be replaced.

The typical execution flow of a game is depicted in Figure 1.3. When a game is started, the game data is first initialized and the corresponding content loaded. The game logic reads the user input, computes the time Δt that has passed since the last frame, updates the game state accordingly and eventually, the game view draws the scene. Since high frame rates are typically attributed to a good gaming experience, most games are programmed such that they execute this game loop as often as possible and without any frame rate control. This in turn leads to a high system utilization and consequently prevents the Linux governor from lowering the frequency.

As it has been shown in [28], frame rates beyond a certain point do not improve the user experience. Hence, game applications that are programmed in the classical infinite

loop approach waste energy by computing frames that actually do not contribute to the user experience. Especially, on mobile devices with small displays this approach can be questioned. In this work, we present a governor that does not only take DVFS decisions based on the system utilization, but is as well *aware* of the gaming application and considers the game's timing and its requirements.

1.3 Power management for games

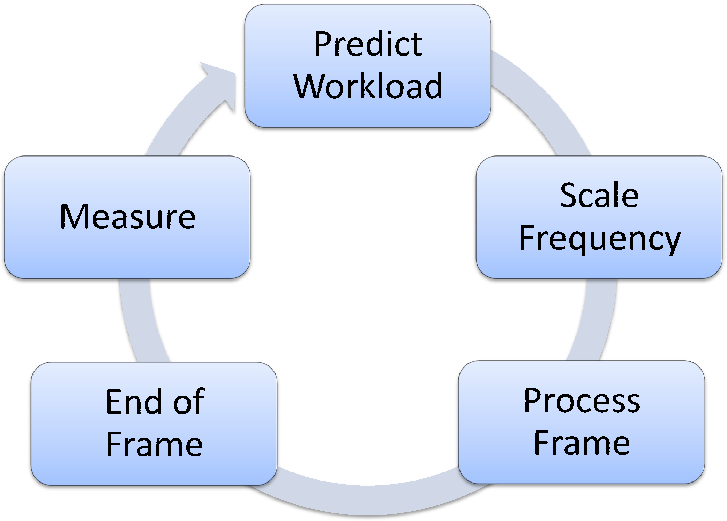


Figure 1.4: Basic structure of the game power management algorithm

As described, games are made up of a sequence of frames. To reduce the power consumption, we measure each frame's processing requirements, predict the future requirement and scale the frequency based on this prediction (see Figure 1.4). Here, the workload $c[i]$ of the i-th frame is measured in terms of processing cycles resulting in

$$c = [c[i],\ c[i-1],\ \ldots,\ c[i-n+1]]^T$$

and past estimation errors

$$e = [e[i],\ e[i-1],\ \ldots,\ e[i-n+1]]^T.$$

Using this data and exploiting statistical relationships, the quantitative prediction of $i+1$-th frame's workload is given by:

$$\tilde{c}[i+1] = predictor(c, e).$$

How exactly these relationships are exploited and what types of predictors are suited will be discussed in Chapter 3 of this work. Based on the prediction and desired frame rate, the required clock frequency of the processor is computed. The processor's frequency is scaled accordingly, the frame is processed (involving AI, physics-related computations,

etc.) and eventually rendered. Finally, we measure the real workload $c[i+1]$ and compute the estimation error

$$e[i+1] = \tilde{c}[i+1] - c[i+1].$$

Both, $c[i+1]$ and $e[i+1]$, are fed back to the workload predictor to forecast the next frame's workload.

While the described scheme can as well be applied to other frame-based applications, there are several differences between games and applications like video decoders which will be described in Section 2.3.4. The general challenges of the approach, as well as the challenges that are specific to games will be detailed in the following.

1.4 Challenges of game power management

The key to efficient power management, but as well one of the main challenges, is an accurate workload prediction. If the future workload is correctly predicted, the processing frequency can be selected such that the energy consumption is optimized under consideration of timing constraints, e.g., defined by a target frame rate. An over-estimation of future workloads, however, wastes energy by running the processor at unnecessarily high processing speeds. On the other hand, if the future workload is under-estimated, the processing of frames might not finish in time which in turn might lead to a deteriorated user experience. Hence, core of the described game power manager is the quantitative forecast of the future workload:

> *"Quantitative forecasting techniques make formal use of historical data and a forecasting model. The model formally summarizes patterns in the data and expresses a statistical relationship between previous and current values of the variable. Then the model is used to project the patterns in the data into the future."* [106]

There is a vast number of forecasting models (predictors), ranging from simple PID controller-based predictors with low computational overhead to complex non-linear prediction techniques. For any kind of workload prediction, the most important requirements to the predictor are as follows:

- **Accuracy**: As discussed, the predictor's accuracy directly impacts the performance of the power management governor and hence should be as high as possible. However, in case of DVFS, the prediction result is quantized by the power manager to one of the finite number of available processing frequencies and therefore prediction errors can be tolerated to a particular extent.
- **Low overhead**: Workload is predicted based on historical data, model parameters and using a prediction algorithm. Since we perform an online prediction on a frame-by-frame basis the computational effort of this prediction should be as low as possible. Determining suitable model parameters can as well be a very time

consuming task, especially in the case of, e.g., complex non-linear models. Here, it has to be differed if the parameters are determined offline or updated using an online self-learning algorithm. For the latter, the algorithm should as well impose only a small overhead, while offline tuning typically is less time critical. In case of very complex offline tuning algorithms it can be considered to offload the tuning to servers and thereby save resources on the mobile platform.

While these two criterias are of great importance for any workload prediction, there are several aspects that are specific to game workload prediction:

- **Robustness**: Specific to games is the fact that game workloads are of interactive and hence very versatile nature, e.g., while an exploration phase of a player walking through a virtual world might only impose small workload variations, within a couple of frames this might change to a highly variable workload due to enemy contact. Hence, the predictor's robustness, i.e., its sensitivity to workload variations to changes in workload characteristics, is of great importance. Ideally, a one-time offline tuning based on recorded sample workloads should be sufficient for the predictor to provide a good experience and optimal power savings throughout the game. This makes it deployable in real setups, where all game plays cannot be known in advance. Robustness as well includes that the predictor should not become unstable due to to large workload variations (see Chapter 3).

- **Flexibility**: Due to the enormous amount of available games, it is desirable that the predictor is not only capable of modeling one game correctly, but as well allows adaptations, e.g., changing parameter values, such that it is applicable to a large variety of games.

- **Portability**: Considering the large number of games entering the market on a daily basis, the solution should not only work for one game, but be portable to many games and even across platforms. In order to predict future processing requirements of a game, the workload needs to be accurately measured on a frame-by-frame basis. In previous work [50–54], id Software's Quake II was used as reference since this game is *open-source* and hence, the code could be directly instrumented. Quake II is from 1997 and based on software rendering, i.e., the graphics are computed by the CPU and without GPU-support. This graphics workload, which amounts up to 90 % of the total workload, typically shows a high inter-frame correlation. As will be shown in this work, this statistical dependency can be exploited in case of Quake II. However, with the advent of powerful GPUs the rendering workload was mostly offloaded in modern games and the freed CPU processing time was used to significantly increase the complexity of physics and AI engines. The physics and AI computations are not necessarily performed on a frame-by-frame basis and hence, the inter-frame correlation of modern game workloads is typically less compared to older games like Quake II. An important question that was left unanswered by previous work is if the techniques that have been successfully applied to Quake II are still applicable to popular modern games. Modern games, however, are *closed-source* and a direct instrumentation of the game's source code is not possible. To

still measure required statistics such as frame timing information, a technique is required to instrument closed-source games or the operating system.

- **Game states**: Specific to games is as well the fact that most games consist of different states, like the level selection, the menu and the level loading states (see Chapter 4). For Android-based games, we observed that up to 50.8 % of the gaming time and hence a significant amount of total energy is not spent in the actual gaming state. Each game state has its individual workload characteristics and requirements in terms of target frame rates. For example, in the level selection menu the frame rate can be reduced due to lower interaction and less animations. To exploit these state-specific requirements and characteristics, the game's current state has to be detected and forwarded to the power manager. Such a detection algorithm should have a low overhead, be applicable to modern closed-source games and again should not be game-specific. Further, for each of the game states a power management strategy has to be found that guarantees an optimal power consumption without negatively impacting the user experience.

In summary, a suitable workload predictor has to be carefully chosen, considering above requirements. If the selection is not done in a systematical manner this might lead to inferior performance or even an unstable prediction, possibly resulting in a high power consumption and a low user satisfaction.

To evaluate the performance of new approaches, measurement results are typically compared to *existing* work to show efficiency of the methods in terms of reduced power consumptions or for example diminished computational impact. An important question often remains unanswered: How much power could be theoretically saved considering an *optimal* algorithm? The knowledge of the gap between existing and optimal algorithms helps to direct future research efforts. There are several challenges one has to overcome in order to answer this question: Games are highly non-deterministic and the impact of changing a parameter, e.g., the processing frequency for a particular frame of a game, can be never compared between two runs of a game since the content and workload will completely differ. This highly complicates deriving frame and cycle-accurate workload models. Further, the processing time of a frame does not only depend on the CPU frequency and the workload in cycles, but is a result of an interplay between the CPU and various components such as the memory, GPU, display and touch sensors. Hence, the workload in terms of CPU cycles does not scale linearly with the processing frequency and an accurate scaling model is required in addition to a game workload model. Identifying and measuring metrics that allow modeling these interdependencies requires a heavy instrumentation of the operating system which in turn might lead to false results due to the instrumentation overhead.

In this thesis, approaches will be presented to systematically answer open questions and develop techniques that help to solve above described challenges. The structure and the contribution of this thesis will be elaborated in the following.

1.5 Contributions and organization

The main contributions of this work are summarized as follows:

- Analysis and identification of real-life game workload prediction techniques
- Development of techniques that allow applying game power management not only to open-source, but as well to closed-source Windows and Android games by DLL injection and library instrumentation respectively
- Development of a smartphone-based measurement setup that allows detailed profiling of closed-source Android games
- Game state detection and game state-specific power management of closed-source games resulting in significant power savings of up to 43.2 % compared to Android's default power manager
- User study-based confirmation that no difference is noticed between the our approach and Android's default power manager
- Theoretical modeling of the optimal power manager and identification of future research directions

The thesis is organized in 6 chapters. This chapter gives an overview, motivates the work and introduces the reader to the basic architecture of games and power management techniques. The remainder of the thesis is structured as follows:

Chapter 2 gives an overview of related work. We discuss general approaches to save power on mobile devices by leveraging processor- and as application-specific techniques. Here, we outline techniques which have been developed for video applications, point out the main differences to gaming applications and show why these techniques cannot directly be applied to games. Existing work in the context of game applications and on power management for peripherals like display, GPU and network interfaces are as well discussed in detail in this chapter.

Accurate and robust workload prediction is the key to efficient power management. In Chapter 3 we evaluate and identify suitable prediction techniques for gaming applications. First, we study PID controller-based predictors which have been successfully applied to video decoding as well as to gaming applications in the past. In previous work, the gain values of the PID-based predictor have been tuned manually and only for one game play of Quake II. We systematically explore the predictor's robustness to variations, observing inferior performance and even an unstable controller if PID gain values, that have been tuned for one game play, are used for different game plays. Due to the highly interactive nature of games and required individual tuning of parameters, the PID controller-based approach turns out to be inappropriate for game workload prediction.
To overcome the need for an individual tuning of parameters, we introduced the least mean squares (LMS) linear predictor which learns its weights automatically and thereby adapts itself to workload changes. We observe a similar prediction performance, but with-

out the need of a per-game play tuning, making the LMS linear predictor a suitable choice for Quake II workload prediction.
All previous work was based on old open-source games like Quake II from 1997, since the game source code had to be modified to gather timing information for the workload prediction. To answer questions towards applicability of developed methods to modern games, which are commonly closed-source, we present a technique that allows collecting frame timings without a direct instrumenting of the game. Using this method, we reveal that modern games like Call of Duty and Crysis show significantly higher workload variations compared to Quake II, resulting in an unstable LMS linear predictor. For both, Quake II as well as modern game workload characteristics, we identify models from the time series analysis domain like the autoregressive moving average (ARMA) to be a perfect choice, guaranteeing a high accuracy, robust prediction and imposing a minimal prediction overhead on the system. The results of this chapter have appeared in [34, 39].

Chapter 4 presents a light-weight graphics instrumentation that allows detecting game states and performing game state-specific power management. Each game consists of several states, like the main menu state, the level menu, the level loading state and the actual gaming state. Each of these states has different workload characteristics and requirements. For example, the main menu is typically less interactive than the gaming state and hence the target frame rate can be significantly reduced and power be saved without affecting the user experience. Further, the loading state of a game commonly is memory-bound, i.e., the CPU is mostly waiting for data. Thus, the CPU processing frequency can be lowered without considerably prolonging loading times. We present an instrumentation technique that allows a reliable detection of different game states based on textures being used in the game. Using this technique, we first analyze the power consumption of individual states using the different governors. Next, we developed a state-specific power management that considers a game state's processing requirement and target frame rate. We show how the power consumption of each individual state can be significantly lowered. We evaluate the time players typically spend in particular game states depending on their playing skills and for three popular Android games, namely Cut the Rope, Temple Run and Jetpack Joyride. For all of the games, we could show that players spend a large amount of time not only in the gaming state, but as well in states like the menu and loading state. In a final comparison we show overall power savings of up to 43.2 % obtained by the `game state specific` governor in comparison to the default Android governor. With the help of a user study we investigate if users notice any difference between the two governors. The work presented in this chapter has been published in [35–38] and received the MobiSys 2013 Best Demo Award.

In Chapter 5, a statistical model is presented that allows answering an important question. How much power could theoretically be saved if the future workload of a game was known? This so-called *oracle* predictor is a theoretical construct and cannot be implemented on a real device. We discuss the difficulties in deriving an oracle model that is accurate on a frame-by-frame basis. Next, we present an accurate statistical model and for the first time compare state-of-the-art game power managers not only to other *existing* power managers like Android's `interactive` governor, but as well to the *optimal* power manager. Based

on these results, we reveal the remaining gap and weaknesses of current power managers and motivate future research in the domain of game workload prediction. In addition, we show that a popular alternative to DVFS, namely race-to-halt, is not a suitable option for the processor used in this study. The results of this chapter have been submitted to IEEE Transactions on Computers in July, 2014.

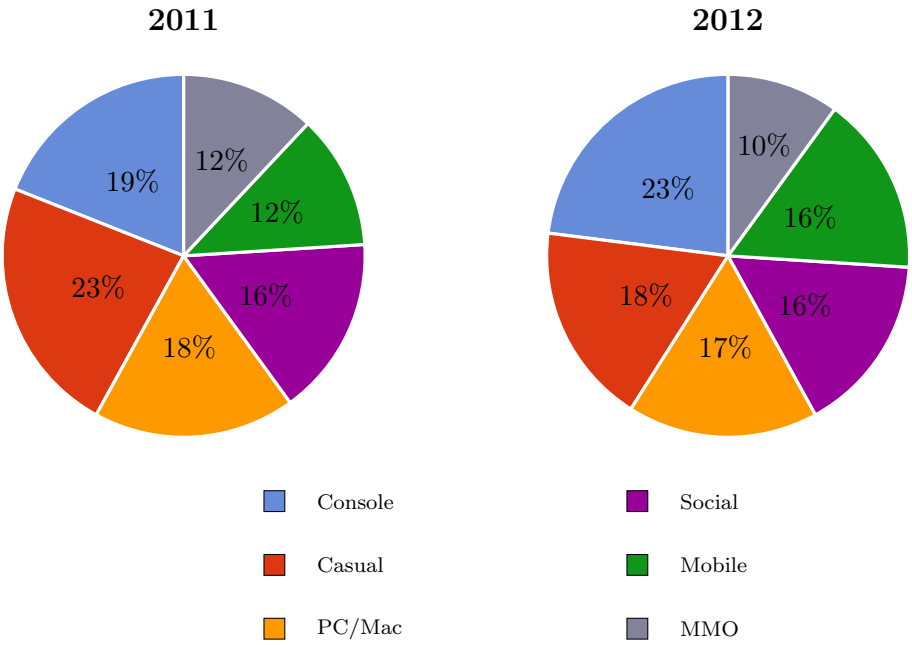


Figure 1.5: Distribution of time spent on video games in the United States in 2011 and 2012 by platform [107, 108]

The distribution of time players spent on video games in the United States by platform is depicted in Figure 1.5 (see [107,108] for more details). The amount of time is almost evenly distributed between console, casual, PC/Mac, Mobile and Massively Multiplayer Online (MMO) games. From 2011 to 2012, the time players spent on mobile devices significantly increased from 12 % to 16 % showing the current trend of smartphone-based gaming. As part of this work, we present three different hardware platforms which will be detailed and discussed in the chapters where they are used. Each of the setups represents a gaming platform that is found in the real world. The software-based rendering setup introduced in Chapter 3 represents cheap mobile devices without any GPU support. Games that are emulated typically do not leverage the hardware-based rendering pipelines and therefore are as well represented by this setup. Most gaming platforms today have a graphics accelerator or a dedicated graphics processing unit. In this work, we present results for both, desktop PC based gaming platforms (see Chapter 3) as well as mobile devices like smartphones or tablet PCs (as used in Chapter 4 and 5). For each of the setups, we detail the software instrumentations and eventual hardware modifications that are required and thereby open the possibility to the research community to develop and test own algorithms and power management strategies.

In Chapter 6 we summarize the results and conclude this work. We motivate future research endeavors towards power efficient gaming and point out promising directions and possible improvements that might be investigated. Note that [37] contains details from all the chapters.

1.6 List of publications

Parts of the contributions presented in this thesis have appeared in the following publications:

- Benedikt Dietrich, Samarjit Chakraborty, *Forget the Battery, Let's Play Games!*, in Symposium on Embedded Systems For Real-time Multimedia (Estimedia), New Delhi, India, 2014.
- Benedikt Dietrich, Samarjit Chakraborty, *Estimating the Limits of CPU Power Management for Mobile Games*, submitted to IEEE Transactions on Computers, July 2014.
- Benedikt Dietrich, Samarjit Chakraborty, *Lightweight Graphics Instrumentation for Game-State specific Power Management in Android*, Multimedia Systems Journal (MMSJ), 20(5):563–578, 2014.
- Benedikt Dietrich, Dip Goswami, Samarjit Chakraborty, Apratim Guha, Matthias Gries, *Time Series Characterization of Gaming Workload for Runtime Power Management*, due to appear in IEEE Transactions on Computers.
- Benedikt Dietrich, Samarjit Chakraborty, *DEMO: Power Management using Game State Detection on Android Smartphones*, in International Conference on Mobile Systems (MobiSys), Taipei, Taiwan, 2013 (Received Best Demo Award).
- Martin Geier, Martin Becker, Daniel Yunge, Benedikt Dietrich, Reinhard Schneider, Dip Goswami, Samarjit Chakraborty, *Let's put the Car in your Phone!*, in Design Automation Conference (DAC), Austin, USA, 2013.
- Benedikt Dietrich, Samarjit Chakraborty, *Managing Power for Closed-Source Android OS Games by Lightweight Graphics Instrumentation*, in NETGAMES, Venice, Italy, 2012.
- Benedikt Dietrich, Swaroop Nunna, Dip Goswami, Samarjit Chakraborty, Matthias Gries, *LMS-based Low-Complexity Game Workload Prediction for DVFS*, in IEEE International Conference on Computer Design (ICCD), Amsterdam, Netherlands, 2010.

2

Related work

Power management is a topic that has received a lot of attention in literature over the past years. In this chapter we provide an overview of work on power management for mobile devices and CPUs in particular. The first part discusses where in a smartphone energy is consumed. We then outline low-power CMOS design techniques that are used in today's mobile processors. Section 2.3 focuses on CPU power management where we describe related work on generic and application-specific DVFS algorithms. Closely related game power management techniques for CPUs are discussed in Section 2.3.4. While DVFS reduces the power consumption of an active CPU, idling power can be reduced using dynamic power management (DPM). Related DPM work is discussed in Section 2.4. The fifth part of this chapter (Section 2.5) presents related work that targets to reduce the power consumption of mobile device peripherals such as the display, memory, GPU and network interface. In this context, we focus on work that has been specifically developed for or can be applied to game applications.

2.1 Smartphone power consumption

The power consumption of a mobile device is composed of the power consumed by all its components such as CPU, Wifi and GSM module, GPU and display. Literature has investigated the main contributors to the total power consumption of mobile devices: [17] states that the display and the GSM module consume most of a smartphone's energy. This

Table 2.1: Power consumption for different applications running on a Samsung Galaxy Nexus. The phone has been modified to measure the CPU and total power consumption separately

Application	CPU [mW]	Total [mW]	CPU power [%]
Facebook	237.8	1226.0	19.4
Temple Run	377.3	1605.9	23.5
Jetpack Joyride	376.3	1481.6	25.4
Cut the Rope	349.2	1449.2	24.1
Shadowgun	467.9	1615.6	28.9

study was performed in 2010 and is based on three, by now, relatively old smartphones: The HTC One, the Openmoko Neo Freerunner and the Google Nexus One. While many people still attribute short battery life-time to the display's high power consumption, for modern smartphones this does not hold true any longer due to advances in display technology and the steady increase of mobile processing power. In a study from 2013, Chen et al. [20] reported for five modern smartphones that the AMOLED displays only consumed between 15 % and 22 % of the total power consumption when games were played. Up to 40 % of the total power was consumed by the CPU. Similar results were obtained on one of our measurement platforms, used in this thesis: While the CPU's share of the total power consumption was only 19.4 % for applications like Facebook, the CPU consumed up to 28.9 % when complex and graphics-intensive games like Shadowgun were played (see Table 2.1). These results clearly show that the CPU is one of the main contributors to the total power consumption. In the following, we give a brief overview on low-level techniques that have been developed to reduce a mobile processor's power consumption.

2.2 Low-power CMOS design

Various techniques have been developed to reduce the static and dynamic power consumption of CMOS circuits. Most important techniques, which as well find application in today's processors, will be discussed in the following. For a more detailed description on the different techniques we would like to refer to [112], while details about their implementation can be found in [77].

Typically, CMOS circuits can be split in critical and less time-critical blocks. *Static voltage scaling* (SVS) leverages this fact and operates the less time-critical blocks, e.g., peripherals of a processor, with a lower supply voltage and frequency, resulting in a reduced dynamic and static power. *Multi-level voltage scaling* (MVS) extends the idea of SVS by providing a choice among two or more operating points (voltage and frequency settings). In case it is dynamically switched between several voltage and frequency levels to follow workload changes, the term *dynamic voltage and frequency* scaling (DVFS) is used [77].

A major contributor to the leakage power, the sub-threshold leakage current can be approximated by:

$$I_{sub} = \mu C_{ox} V_\theta^2 \frac{W}{L} e^{\frac{V_{GS}-V_T}{nV_\theta}}, \tag{2.1}$$

where μ is the carrier mobility, C_{ox} is the gate capacitance, V_θ is the thermal voltage given by kT/q (25.9 mV at room temperature), W and L are gate width and length respectively, V_{GS} is the gate-source voltage and n is a function of the device fabrication process (ranging from 1.0 to 2.5) [77]. The threshold voltage V_T is the gate-source voltage that is at least required to allow a current flow between source and drain of a MOS transistor. The leakage current depends exponentially on the difference between V_{GS} and V_T. MOS devices with higher V_T have lower leakage current, but are slower since a higher V_{GS} is required for switching. To reduce the power consumption, circuits are built with different MOS devices: Less time-critical parts are built using MOS devices with a larger V_T, whereas performance critical blocks are built with low-V_T MOS devices. This design method is commonly referred to as *multiple threshold voltage* design.
Another circuit level optimization is the use of *long-channel devices*. According to Equation (2.1) the sub-threshold leakage current is proportional to $1/L$, where L is the gate length of the transistor. Hence, by using devices with longer gates the leakage current can be reduced. However, longer gates are less performant, require more space and have an increased dynamic power consumption. A careful optimization has to be performed to successfully reduce the total power consumption using this technique [77].
Adaptive body biasing allows changing the threshold voltage dynamically, by applying a reverse bias to the NMOS/PMOS bodies. For example, by applying a voltage to the NMOS body that is lower than ground, the threshold voltage can be increased and leakage current decreased. Thereby, circuit blocks can be put into low leakage modes when idling [76].
As discussed in Chapter 1, *clock gating* reduces the dynamic power consumption of a circuit by cutting off the clock supply to parts of the circuit that are currently not required [116, 143, 155]. *Power gating* cuts off the supply voltage, thereby reducing both, the static and the dynamic power consumption. While power gating was first introduced to dynamically resize caches [122], it was soon used to put other functional units of a processor to sleep [27, 43, 64, 126]. Both, clock and power gating are used for DPM.

Using DPM and DVFS, developers can dynamically change the state of a system to reduce its power consumption according to workload characteristics. In the following we first discuss DVFS to later present related work on DPM.

2.3 Dynamic voltage and frequency scaling

Most of today's processors provide the possibility to dynamically scale the voltage and frequency. Instead of running the processor always at the maximum frequency, the processing speed is adapted to the requirements of applications being executed on the processor. In

the following we first discuss choices that have to be made during the design time of the processor to provide efficient DVFS.

2.3.1 Design time considerations

Ideal DVFS assumes that a continuous scaling of operating voltage and frequency is instantaneously possible [26]. In real circuits, however, the number of voltage and frequency levels (operating points) is limited to a discrete number due to additionally required hardware causing increased design complexity, cost, area, delay and power penalties [65]. The decision about the exact operating points, which should be offered by the circuit to optimize the energy consumption, has to be made during design time of the circuit and is referred to as voltage setup problem. The minimum possible supply voltage V_{dd} of a CMOS circuit is investigated in [102]. It is shown that V_{dd} can even be decreased below the threshold voltage V_T. Zhai et al. [160] showed that a decrease to this extend is only beneficial in rare cases since circuit delays are growing exponentially if $V_{dd} < V_T$ and consequently leakage power starts dominating the circuit's total power consumption. Hua and Qu [65] propose a method for finding the optimal operating points for a dual-voltage and an analytical approach for the multi-voltage level design. It is shown, that the practical multiple-voltage DVFS can reach the full potential of an ideal DVFS system, assuming immediate voltage and frequency transitions. The work is extended in [157] by considering the transition overhead.

Once operating points of a circuit are defined, hardware designers typically add safety margins to avoid timing errors due to variations in the manufacturing process. To avoid an increase of power consumption due to these margins, in [31] an automatic post-production algorithm is suggested to determine minimum possible operating voltages for each frequency level and the chip at hands: The operating voltage is automatically reduced until the point of failure which is checked by an error detection and correction mechanism. Thereby, safety margins are eliminated and operating points are optimized. All of the processors used in this work provide a fixed number of operating frequency levels. However, in case of the OMAP4460 mobile processor (see Chapter 4) the voltage can be set for each individual frequency level in the corresponding kernel module, allowing a voltage optimization for the processor at hands. Towards this, the operating voltage has to be step-wise reduced as suggested in [31] and the functionality checked leading to the lowest valid voltage setting for the specific frequency. Since we want our work to be comparable and experiments to be repeatable we decided against this chip-specific tuning and used the default Android settings.

2.3.2 Run-time implications of DVFS

The choice of operating points during design time leads to a set of operating points among which the power manager can choose to adapt the processor speed according to the future workload. In many scenarios, tasks being executed on a processor, need to finish within a

given hard or soft deadline. In order to guarantee that no deadlines are violated it needs to be known how long a task will take at the different processing frequencies.

Relative performance

Assuming that the workload C of a task in cycles is known or can be accurately predicted, one simple assumption is that the execution time t linearly scales with the processing frequency:

$$t(f) = \frac{C}{f}. \tag{2.2}$$

However, the computational workload C does not only include on-chip CPU computations, but as well cycles during which the CPU has to wait for components like the memory or bus operations. In [24–26, 138] the workload of a task is partitioned into on- and off-chip workload. The on-chip workload scales linearly with the CPU frequency, while the off-chip workload is independent of the scaling:

$$t(f_{CPU}) = \frac{C_{on-chip}}{f_{CPU}} + \frac{C_{memory}}{f_{memory}} + \frac{C_{bus}}{f_{bus}} + \dots$$

It is shown that the ratio of on- to off-chip workload can be approximated by performance counters of the CPU. These counters can be configured to gather statistics about cache misses, number of instructions being executed, ALU operations, etc. and therefore reflect a tasks's workload composition. Snowdon et al. [138] introduced the so-called relative performance, which describes the CPU-boundedness of a workload and is defined by

$$s(f) = \frac{f_{max}}{f} \times \frac{c(f_{max})}{c(f)} = \frac{t(f_{max})}{t(f)}, \tag{2.3}$$

where f_{max} is the maximum available CPU frequency, f is a valid operating frequency of the processor, $c(f)$ is the number of cycles and $t(f)$ the amount of time required at frequency f. Snowdon et al. [138] as well leverage the CPU performance counters to compute the relative performance of currently running tasks. The impact on future workloads is then predicted assuming a strong correlation between the past and near-future relative performance. Based on this relative performance and Equation (2.3), the required frequency to finish workload C in time can be computed more accurately compared to the pessimistic assumption in Equation (2.2).

Above work is based on benchmark applications without the need of user interaction or graphical interfaces. Boundedness by GPU or user I/O, which is very likely for gaming applications, is not considered. In Chapter 5 we elaborate why considering this impact is very difficult in case of games.

Critical speed

The choice of CPU frequency does not only influence the processing time, but as well the total energy consumption. In general, it cannot be assumed that the slowest operating point will always yield maximum energy savings for the completion of a task running on the processor: The total energy consumed by a computing system is composed of the CPU's power consumption and the power consumed by other devices like memory or GPU. While reducing the CPU's frequency lowers the processor's power consumption, other components will still consume the same power over a possibly prolonged execution time. Thereby, the total energy consumed for completion of a task might even increase by lowering the CPU frequency.

The *critical speed* is defined as the frequency that minimizes the system's energy consumption for a given task and considering described factors [75]. The work considers a set of periodic real-time tasks which are scheduled using earliest deadline first (EDF) scheduling and whose deadlines, power consumption and slowdown factors at each frequency are known. For this setup it is shown that the power consumption can be significantly decreased if the critical speed of tasks is considered. In [137] it is shown that this holds as well true for real benchmarks running on a PXA255-based measurement setup. Further, the critical speed's dependency on the type of workload is demonstrated: For the given system and memory-bound workloads the critical speed was observed to be significantly lower than for CPU-bound workloads. Lowering the processing speed for CPU-bound workloads will significantly increase the processing time and thereby as well the static energy consumed during the prolonged processing time. The execution of memory-bound workloads, in contrast, is nearly independent of the CPU speed and hence always a similar amount of static energy is consumed while the dynamic energy can be significantly lowered by running the processor at a low processing speed.

In [86] a Linux-based power management governor has been developed that considers the critical speed and memory-boundedness of running tasks. Towards this, the memory-access rate (MAR) is computed, based on the number of instruction and data cache misses per instruction. The governor then scales the processor's frequency based on the utilization under consideration of the current MAR and critical speed. An Android-based implementation of this governor is presented in [87]. This governor targets to optimize the energy consumption for a given task without considering timing constraints. As described in Chapter 1 this leads to a high utilization and energy consumption in the case of games. In addition, the governor only considers the waiting time for memory. It has to be taken into consideration that graphics intensive games can as well be I/O- or GPU-bound.

In this work, we investigate implications of the slow down and show in Chapter 4 that for all games used in this study the total energy consumption of the smartphone continuously decreases with the CPU frequency. The critical speed was always observed to equal to the lowest available frequency.

To successfully apply DVFS on a processor the future workload has to be known in advance such that the correct processing frequency can be chosen. Otherwise, energy is

wasted by running the processor at too high processing frequencies or tasks finish not in time if too low frequencies are used. For hard real-time systems it is often assumed that workloads, deadlines, periods and energy consumptions of tasks are known, forming an offline optimization problem and resulting in a static schedule [62, 74, 123, 156].

On real platforms like smartphones, in contrast, characteristics of future workloads cannot be known and need to be predicted. Existing work can be divided into generic and application-specific methods. In the following, we are going to discuss generic approaches which only leverage information that is provided by the OS and hardware metrics. Application-specific approaches are then discussed in Section 2.3.4.

2.3.3 Generic DVFS

The first generic operating system-based solution to improve the energy consumption has been proposed by Weiser et al. [153]. Here, the utilization of the processor, i.e., the percentage of time the processor is idling, is monitored on an interval-basis. If the utilization exceeds an upper threshold during the last interval, the power manager increases the processing frequency, while the frequency is lowered in case the utilization drops below a lower threshold. In [49] this approach is compared to other schemes, for example leveraging a weighted average of past utilizations as prediction. As detailed in Chapter 1 and [110], the utilization-based approach is still used in the Linux power manager. The success of this type of power manager can be explained by its simplicity, which allows easy porting to various platforms, an important aspect for a Linux Kernel Module (LKM). Main drawback of these kind of power managers is that they are completely unaware of the running applications and their requirements.

In [24–26,138,139] it is shown that the performance of power managers can be significantly increased if effects like memory-boundness are considered by decomposing workloads in on- and off-chip workloads as described in Section 2.3.2. The decomposition is performed based on performance counter values and a very fine-grained offline analysis of measurement data. Performance counters that allow a decomposition first have to be identified to then tune linear models describing the decomposition. In [32] the workload is decomposed based on cycles per instruction (CPI) used by the CPU and CPI of caches and stalls. To estimate the future ratio between these two, an online learning algorithm is used and thereby the offline analysis and tuning required in previously mentioned work is avoided. Similarly, in [72, 73] workload phases and durations are predicted by exploiting statistical correlations between the past and near future. Workload phase, here refers to an interval during which, e.g., the instruction per cycle (IPC) count is below average. Predicting such phases and their durations allows the power manager to take decisions if a frequency transition is beneficial under consideration of the transition overhead and the next phase's duration. Experiments are based on 25 single-threaded SPEC2000 benchmarks exposing workload phase durations of up to several seconds. Applicability of presented methods to gaming applications can be questioned since games are typically highly mutli-threaded. The scheduler switches rapidly between different threads and therefore fast phase changes are likely. Another drawback of all above described performance counter based methods

is that the implementation is highly processor and architecture specific. The models need to be tuned to determine correlations between performance counter readings and future workload characteristics. This tuning is time consuming and makes proposed solutions less attractive.

Discussed power managers have the advantage of being generically applicable (once processor specific parameters are identified) making them a good choice for operating system-based power management. Compared to running the processor at highest processing speed, significant power savings can be obtained with neglectable performance implications. For in Chapter 1 discussed reasons these utilization-based approaches are not efficient for gaming applications. As will be shown in the following, knowledge about running applications typically allows developers to further improve power managers by exploiting application-specific details.

2.3.4 Application-aware DVFS

Various approaches have been suggested in the past to improve power managers by making them aware of running applications. In this context, the a class of applications that has been studied most are video applications.

Video applications

Most common video codecs like H.264 encode image information using three different frame types: I-, P- and B-frames. The I-frame can be decoded on its own, while for a P-frame information from previous I- or P-frames is required. Decoding a B-frame requires information from surrounding I- and P-frames. The resulting encoded video stream consists of a repeating sequence (named group of pictures) of these frames, e.g., I-B-B-P-B-B-P-B-B.

To decode such a stream, several tasks have to be performed on a frame basis: Variable length decoding (VLD), inverse discrete cosine transformation (IDCT) and motion compensation (MC). The exact composition of the workload depends on the type and the content of the frame. Frames should be decoded at a particular playout rate to provide a good user experience without stuttering in the playback. Hence, if DVFS is applied, application-aware governors should choose the processing frequency such that the three tasks are processed in time for each frame guaranteeing the desired playout rate. Towards this, an accurate workload prediction is required: If the workload is under-predicted, frames might not be decoded in time, resulting in a deteroriated user experience due to stuttering. Over-prediction, on the contrary, wastes energy since the frames could have been decoded with a lower frequency. Sufficient workload variability is one important premise for the applicability of DVFS and has been shown for H.263 decoders in [67, 121]. Various methods have been discussed in literature to exploit this variability and reduce the energy consumption while preserving the quality based on an accurate video workload prediction.

History-based approaches assume correlation between decoding workloads of consecutive frames: Frame content usually changes slowly unless of scene changes happening. In [68] the instruction count (IC) is predicted for each frame based on the maximum IC of the last 5 frames of the same type. Based on the instructions per cycle (IPC) of the last frame of the same type and the predicted IC, the workload is predicted and the frequency chosen. Choi et al. [23] divide the workload of a MPEG decoder into two parts: The frame dependent and the frame independent workload. The frame dependent workload is predicted based on three moving averages over past frame workloads (for each frame type one moving average). The frame independent workload is constant and, in case a misprediction was made during the frame dependent part, the timing is corrected by scaling the frequency again for the frame independent workload such that the misprediction is corrected. The scheme presented in [118] filters past variations using an exponential weighted moving average of previous frames as prediction. In [93,158] the future frequency is predicted using probability distributions of workloads. While in [93] these distributions are computed using offline analysis of tracing data, in [158] they are dynamically generated during run-time. In the latter work, multiple tasks are supported by integrating scheduling and DVFS. Both approaches outperform the generic utilization-based prediction presented in [153]. In this thesis we develop and evaluate DVFS algorithms for games. Similar to videos, games are frame based and their workload shows significant variations making them amenable to DVFS. The applicability of statistical approaches as discussed above will be in detail elaborated in Chapter 3 of this thesis.

Offline watermarking-based methods typically have a higher accuracy and lower prediction and instrumentation overheads compared to history-based approaches. In [121] a H.263 video encoder is changed to annotate frames with complexity information. A strong correlation between the frame size, type and the decoding time is leveraged by simply providing the frame size as metadata. Based on the frame size metadata and three linear regression models (one for each frame type), the frame decoding time could be accurately predicted to apply DVFS on the decoder side. This approach outperformed the static fixed-frequency as well as the utilization-based power manager. As described above, the workload of a MPEG-2 decoder can be split in three main tasks: VLD, IDCT and MC. Huang et al. [66] demonstrate that for each of the tasks the workload can be predicted based on characteristics of the stream, for example exploiting the strong linear correlation between the VLD task's workload and the number of non-zero coefficients in a stream. For all three tasks, accurate models are found, videos are analyzed and watermarked with corresponding information by a fast transcoder, before being streamed. The decoder extracts the information, predicts the workload based on developed models and scales the frequency accordingly. Main advantage of this approach over [121] is the fact that the encoder does not need to be changed. The need to change the decoder and pre-analyze videos is one of the main drawbacks of offline watermarking based approaches. Watermarking is not applicable to games since the game content is dynamically generated based on the user input and hence cannot be annotated in advance.

While videos and games are both frame based, they are different in several aspects: Videos share a basic underlying structural information that can be exploited which is not the

case for most games. Games are interactive which means that the content and workload characteristics depend on the user. A professional user might run through levels quickly imposing a possibly larger workload on the CPU compared to a beginner player who slowly explores levels. Hence, the power manager might have to adapt its settings in accordance to the person playing which is not the case for videos. Further, due to the interactive nature, game frames cannot be buffered or pre-computed as it is possible for video frames. While most modern games are closed-source, many video decoders are open-source and can be easily instrumented to provide information to the power manager. These differences complicate the direct applicability of methods that have been developed for videos. In the following we describe work on game power management that is closely related to this thesis.

2.3.5 DVFS for game applications

There is only little work that focuses on reducing the processor's power consumption while games are being played. In [88, 97] a user-driven voltage and frequency scaling is presented. As long as the user does not provide negative feedback to the power manager using keyboard shortcuts, the operating frequency is steadily decreased at a particular decrease rate (initially at a 10 seconds interval). Negative feedback will increase the current operating frequency and update the decrease rate of the algorithm. The approach was evaluated using a group of 20 users and based on the soccer game FIFA. Power savings of 22.1 % compared to Windows XP's power manager are reported. Considering the large workload variations that can be observed for modern game applications an update rate of 10 seconds appears suboptimal and might lead to a deteriorated user experience and non-optimal power savings. Further, frequently required user feedback appears highly unattractive for action-rich gaming applications and is likely to annoy the user.

The work of Yan Gu [51–54] focuses on reducing the power consumption of games, using Quake II as a reference game. The interested reader is referred to the full version of Yan Gu's thesis [50]. In [54] it is shown that the frame workload of Quake II exhibits significant variations making the game amenable to DVFS. In [52] it is shown that substantial power savings can be obtained if a PID controller-based predictor is used to forecast future workloads. A manual tuning of the PID controller gain values is suggested. This work left open questions about the robustness of the suggested predictor and the applicability to modern games (Quake II was released in 1997). A detailed discussion of mentioned work is presented in Chapter 3 of this work. There we will discuss related work, analyze the approach in detail, point out weaknesses and suggest a better suited and generally applicable solution to game workload prediction.

As discussed in Chapter 1, the game workload in general is composed of AI, physics and rendering computations. In case of Quake II, the rendering, which is performed in software and without GPU support, constitutes the largest part to the total workload. As shown in [53], this workload is composed of 5 different workloads of which each can be approximated as follows:

- **Brush models** are used to construct the world of Quake II, e.g., to draw solids like walls, floors and ceiling. The rasterization workload of brush models strongly correlates with the number of polygons of the brush model. Hence, once the number of polygons is known, the workload can be approximated.
- **Alias models** are used to rasterize entities like monsters, soldiers, weapons and collectible items, all composed of triangles. The workload can be predicted from the number of pixels of the alias models' triangles.
- **Textures** are graphics applied to brush models to give a realistic impression of bricks, floor, etc.. The texturing workload is with up to 65 % the main contributor to the total rasterization workload. It can be approximated using the number of surfaces in textures. Since computing the number of surfaces in advance requires rasterizing the brush models which is costly, the texture workload is approximated using a history-based prediction.
- Rasterizing **Light** in Quake II is performed using pre-computed light maps that are rendered as multiple surfaces. This rasterization workload is already included in the texture workload.
- **Particles** are used to animate blood or splintering caused by gun shots. Again, the workload can be approximated by the number of constituting pixels.

Using above relations, the workload of each individual object in the view frustum of the user can be computed. Accordingly, the total workload is approximated in [53] from the workload sum of all objects. Main advantage of this game structure-based approach is the accuracy that can be obtained using above described models. However, there are several drawbacks of this method: Predicting the workload from structural information comes with a large computational overhead, since a lot of information has to be pre-computed to be able to evaluate the models. This problem has been tackled in [51] where a hybrid scheme is presented. During phases with low variation, the control-theoretic approach is used since it imposes a negligible overhead and even outperforms the structural approach in terms of prediction performance. For game phases with high variations, the structural outperforms the control-theoretic approach, but comes at the cost of a significantly larger overhead. The hybrid scheme switches between both approaches, depending on the current workload variation phase, thereby optimizing the overhead and prediction error at the same time. The greatest disadvantage of the structural and hybrid approach is that both are highly optimized for one specific game. In order to port the approach to a new game, first the game's source code has to be heavily instrumented. Second, workload models have to be derived which is a highly time consuming task. Most modern games are closed-source and hence this is only possible for the game developers. This, in turn, makes the solution unattractive since the gaming market is of a rapidly changing and highly competitive nature with short time to market intervals. Further, it is not clear, if methods that have been evaluated using Quake II from 1997, are still viable for modern games with hardware-based rendering. In this work, we avoid leveraging information that requires a direct instrumentation of the game's source code. Thereby, applicability and portability are guaranteed even for modern closed-source games.

DVFS reduces the power consumption of active CPUs. In the following we present related work on DPM which reduces the power consumption of idling components and processing cores.

2.4 Dynamic power management

Dynamic power management (DPM) dynamically puts circuit blocks, e.g., idling components of a system-on-chip (SoC), into low-power states and wakes them up again once they are required. Switching between different power states consumes energy and time during which the component is not operable. Hence, a greedy switching to low-power states once a component is idling, is not a viable option. Towards optimal power savings, well-grounded decisions have to be made about when and to which state it is worth to switch. In [14] several prediction techniques are surveyed that have been discussed in literature in the past. It is shown that simple timeout mechanisms, commonly used for laptop displays or harddrives, waste energy by waiting for the expiration of timeouts. Predictive shutdown and wakeup policies try to solve this by forecasting future idle times and activity events based on observations of the past. Further, Markov chain-based schemes are introduced that model stochastic properties of the system, providing the possibility to choose among not only one, but several idle states.

While [14] focuses on system components like harddrives, same techniques can be applied to the CPU. As discussed in Chapter 1 most modern CPUs provide several C-states, i.e., idling states of different depths. In Linux the `cpuidle` module takes care of switching between C-states. The structure, interfaces and governors of this module are described in [111]. Most interesting to this work is the `menu` governor which is the default governor used in the Android setup of this work. The `menu` governor analyzes expected parameters like sleep times, latency requirements, etc. and switches to the most beneficial C-state [111]. In this work, we as well rely on the Linux default `menu` governor for idle state management.

Due to transition costs it is important to keep the processor in the idle state as long as possible. In Linux the scheduler is typically executed periodically at a fixed rate, namely the tick rate. Hence, idling CPUs are woken up periodically resulting in short idle times and an increased power consumption. To avoid these periodic wake-ups, the tickless Kernel has been introduced [48,131,140]: In case the next scheduled timer event is further away than the next tick, the tick interrupt is changed to that of the next timer event. Thereby, unnecessary periodic interrupts are avoided and the CPU can stay longer in the idle state. The Linux Kernel versions used for the Android-based hardware platforms of this thesis are tickless.

A popular alternative to above described DVFS techniques that leverages DPM is *race-to-halt*. Slack is defined as the processing time not used in a system. While the traditional DVFS chooses a frequency that minimizes the slack between the processing time of a task and its deadline, race-to-halt executes the task at the maximum frequency to maximize

the slack. Once the task is processed, the CPU is put into a sleep state where it remains until the next task arrives. This makes a workload prediction redundant and at the same time guarantees that the task, if possible, finishes in time. In [141] the effectiveness of DVFS and sleep states is analyzed for server, desktop and mobile processors. Experiments are based on Java benchmarks, MPEG playback and Apache-based web-page servicing and show that a combination of both, DVFS and idle states is most efficient, especially for workloads where the CPU is under-utilized. As will be shown in Chapter 5, the power consumption of race-to-halt is significantly higher compared to traditional DVFS when being applied to games.

2.5 Power management for peripherals

While the CPU is one of the main power consumers in modern mobile devices, there are several other peripherals such as the display and GPU which as well consume a substantial amount of power. In the following we present related work that focuses on reducing the power consumption of such components. We restrict the overview to work that has been done or is applicable in the context of games. All of the in the following discussed work is orthogonal to the CPU power management techniques presented in this work. Hence, described techniques can be combined with our schemes to obtain an overall energy-optimized system.

Wireless network

The work presented in [6,7,149] focuses on reducing the power consumption of the wireless communication of online multiplayer games. In [6] several approaches are evaluated to reduce the power consumption of the wireless network interface card (WNIC) during game plays of Quake II. First, it is shown that a simple reduction of the data rate is not beneficial in terms of power due to the small packet size of Quake II. The second approach switches off the WNIC whenever the player's interaction level with the game is low. In addition, the player's actions (e.g., hiding, walking, shooting) are classified during the game play and considered in the decision when to switch off the WNIC. It has been shown that the power consumed by the WNIC can be reduced by up to 21 %. Since the classification of the player's action requires a heavy instrumentation of the game's source code, which in most cases is not publicly available, [7] proposes an API, allowing game programmers to report positions of interactive objects (e.g., enemies) and their vision range. This data is used by the resource controller running on the server to compute the velocity and direction of the objects. Further, current game states and their importance for the game play can be reported. Based on this data, the server decides about power state transitions of the clients' wireless modules. This work is extended in [149] with a technique that looks ahead and predicts the player's future position, allowing to power down the WNIC more efficiently.

Display

Another main contributor to the total power consumption of mobile devices is the display. Today, two different types of display technologies are commonly used in smartphones and tablet PCs: LCD panels and OLED displays. While LCD panels are cheap, their main drawback is the need for bright backlight illumination which consumes a significant amount of power. One popular technique to reduce backlight power is based on increasing the luminance of the image to be displayed while dimming the LCD backlight accordingly [18, 22, 70, 71, 85, 129]. In [8] it has been shown that this technique can be successfully applied to highly dynamic first person shooter games like Quake III without impacting the by the user perceived quality.

Especially in the high-end smartphone market, OLED technology started to replace LCDs. OLED displays consume significantly less power, have an increased brightness level and provide possibilities for thinner form factors, but are more expensive than LCD panels. The total power consumption of an OLED display is given by the sum of power consumed by each pixel. In [142] parts of the OLED display are dimmed which are currently not in the focus of the user, for example the lower part of a news article. While this approach is applicable to applications with low update rates like news reading or browsing, the user experience will be clearly affected in the case of highly dynamic game applications during which the user changes his focus quickly. Similar to backlight scaling of LCDs, the supply voltage of OLED displays can be scaled down to reduce the power loss. To compensate reduced luminance the image data has to be modified. In [130] it has been shown that this method is applicable to still images without visible distortion. This work has been extended in [21] to make it applicable to video streaming applications. Studies on the applicability of this method on games have not been performed, yet.

In [90] it has been shown that not all pixel value changes are perceivable by the human visual system (HVS). Further, not all regions of images receive the same amount of attention. This is exploited in the work of [89] where pixel values are changed to reduce the total power consumption of OLED displays, but without impacting the perceived quality. In [42, 125] it is shown that significant amount of power can be saved if user interface colors are chosen under consideration of the displays power consumption.

The display reads and updates its content at a particular rate, namely the refresh rate. On Android devices, this rate is configured to be constant (typically at 60 Hz). In [80] it is shown that for most applications the content does not change that quickly. A scheme is proposed that changes the refresh rate in accordance with the content rate, i.e., the rate at which the displayed content really changes. Thereby, unnecessary display updates are avoided and significant amount of energy is saved. Similar approaches are followed by NVIDIA's G-Sync technology [109] and the recently standardized VESA Adaptive-Sync [152]. While the first goal was to lower display energy required for refreshes, the adaptive synchronization improved as well display effects like tearing and stuttering. The game power management scheme introduced in Chapter 1 maintains constant target frame rates during different phases of the game and hence it could directly adapt the display refresh rate.

GPU

To enable advanced 3D graphics on mobile devices, GPU architectures have been highly optimized to reduce their power consumption. A popular alternative to the traditional immediate mode rendering followed by NVIDIA and AMD, is a technique called deferred (or tiled) rendering found in Imagination Technologies' and Qualcomm's Adreno GPUs. This technique targets to minimize GPU off-chip memory accesses which are expensive in terms of time and energy. Further, the so-called overdraw is reduced, i.e., computations for vertices or pixels which are not visible in the final picture due to occlusion by objects in front. Main drawback of the technique over the traditional approach is an increased rendering latency and hardware complexity. We would like to refer to [4, 69] for more details.

Besides architectural modifications from the manufacturer's side several techniques have been proposed in literature to further reduce the GPU's power consumption. In [46] HVS characteristics are exploited to dynamically switch between Gouraud and Phong shading algorithms whenever the graphics content allows it. While Phong provides preciser shading it consumes significantly more power and is not required for highly dynamic scenes. It is shown that instead the low-precision and less-power hungry Gouraud shading algorithm can be used. Thereby the overall power consumption of the shading stage could be significantly reduced. Hosseini et al. [63] suggest omitting selected lighting effects to reduce the power consumption. While in this work the effects are omitted for all objects a selective application to objects with higher relevance is suggested for the future. The thesis of Pool [120] focuses on reducing the energy consumption by reducing the computational precision of graphics processing stages. In addition, methods are presented to enhance current compression algorithms and thereby reduce the off-chip bandwidth and resulting energy consumption.

Another possibility to reduce the GPU's power consumption is the application of dynamic voltage and frequency scaling. In [104] an OpenGL ES software library is executed on the target platform to gather traces of individual stages of the graphics pipeline. Based on these traces the performance and the power consumption are then analyzed using an ARM instruction-level simulator. Results indicate that the graphics workload exposes sufficient variations making the different stages of the GPU amenable to DVFS. It is shown that already a simple history-based predictor obtains significant power savings for the used simulation setup with tolerable prediction errors. The work motivates specialized prediction techniques to avoid performance impacts due to miss-predictions of the coarse history-based predictor. In [105] a signature-based prediction of GPU workloads is suggested. Signatures are created based on the average triangle area, count and height as well as the vertex count. For each signature the workload is recorded and an entry to a signature-workload table is made. If a similar signature is found in the table the corresponding workload is used as prediction for dynamic voltage and frequency scaling. Both of the above described works are based on ARM performance and power models. The validity of these models can be questioned since graphics pipeline are highly parallelized and optimized and hence are likely to behave very different from the used ARM

processing core models. The work presented in [132] shows that the high workload correlation between tiles of consecutive frames can be effectively exploited to apply DVFS for deferred rendering architectures. In addition to a simple tile-history based predictor, so-called tile ranks are computed based on information extracted from the geometry and tiling stage. The rank indicates whether the workload of the current tile increased, decreased or approximately remained the same in comparison to the previous frame. If the rank of the current tile deviates by more than a particular threshold from the same tile of the previous frame, it is rendered with the maximum frequency to avoid that the tile misses its deadline. Using this technique and dynamically correcting miss-prediction errors by recomputing the remaining slack, the number of frames missing their deadline was minimized while significant power savings could be shown. The results of this work are obtained with the ATTILA simulator [13] which has been modified to simulate a deferred rendering architecture.

Ma et al. [95] investigated the performance bottleneck and the power consumption of the graphics pipeline when games are being played on three different smartphones. Towards this, the pipeline was logically separated into 5 stages: The application stage, including all work performed by the CPU, the geometry stage, computing vertex attributes and 3D transformations, texture fetching, fragment shading, executing pixel shaders and pixel processing, responsible for post-fragment shading pixel processing and writing of color, depth, stencil buffers and alpha blending. By disabling individual stages directly in the Quake III and XRacer source code, differences in terms of performance and consumed power could be directly measured on the smartphones. It is shown that the geometry stage requires most power (up to 35 % of the GPU's total power) and processing time (up to 40 %). Further, it is pointed out that in case of mobile devices the game logic being executed on the CPU consumes significantly more power than the same logic being executed on a desktop architecture (only 15.12 %). While for Quake III the GPU consumed more power than the CPU, this was observed to be the opposite for the game XRacer, highly motivating CPU power management.

The work of Pathania et al. [115] suggests an integrated CPU-GPU power management for 3D mobile games. First, the relationship between different frequency settings, the resulting power consumption and frame rate is analyzed for the game Asphalt 7: Heat. Similar to our findings presented in Chapter 4, it is shown that the game has typical phases during which it is CPU- or GPU-bound. An algorithm is developed that targets to maintain a frame rate within a particular hysteresis (e.g., 30-35 FPS) over a sliding window of 5 seconds. Towards this, the algorithm first starts at the lowest CPU and GPU frequency and extrapolates, based on the current CPU and GPU costs, what frequency is required to maintain the target FPS. This is repeated until the target frame rate is achieved. Since it has been observed that the frame rate is highly sensitive to the GPU frequency, in this state only the CPU frequency is adapted to guarantee that the FPS stays within the defined hysteresis. During scene transitions, the CPU frequency is raised to a maximum, while the GPU frequency is lowered to the minimum possible frequency. The proposed scheme is purely reactive: The frame rate is averaged over a 5 seconds interval and once a drop of this averaged frame rate is observed, the proposed

scheme reacts. For the games we used in this work, we observed that the workload can significantly vary on a very fine-grained frame to frame-basis. Coarsely averaging over an interval of 5 seconds would result in a high percentage of frames missing their deadline and consequently in a deteriorated user experience. To avoid frame rate drops at the first place, the power manager proposed in this work is based on a workload predictor. The prediction and scaling is performed on a frame-by-frame basis, guaranteeing that already small frame workload variations are quickly detected and compensated by scaling the processor's frequency.

Memory

While a significant amount of work focuses on techniques to lower the memory system's power consumption [40, 56, 60, 96, 112, 133], only little work investigated memory power management for in the context of mobile computing. In general, memory is considered as a minor contributor to the total power consumption of a smartphone. In [81] it has been shown that a smartphone's memory can have significant performance implications and might prolong loading times for interactive applications like web browsing. According to the study, slower storages not only increase the I/O-waiting time of the CPU, but as well the duration during which the CPU is active and thereby might have a direct impact on the processor's power consumption. As part of future work, it would be interesting to investigate implications of different memories on the performance in the context of games. In [44] a hybrid memory system for mobile devices is proposed that consists of Mobile RAM and Phase Change Memory (PCM). While Mobile RAM has a lower access latency it requires significant amount of energy for refreshes during idle states. PCM on the contrary has longer access times, but does not require refreshes while idling. In proposed work, the advantages of both memory types are exploited by keeping frequently accessed applications in the Mobile RAM and the others in PCM. For the hardware platforms considered in this work, the memory sub-system cannot be changed and hence is considered as given.

2.6 Summary

In this chapter, we have discussed work that is related to this thesis. We first illustrated a mobile phone's architecture and showed that the processor is one of the main contributors to total power consumption. We presented low-level techniques which have been developed and are nowadays common in mobile processors to lower their power consumption. Related work of two famous approaches which use these low-level techniques, namely DVFS and DPM, have been discussed in detail. We pointed out why approaches that have been developed for video applications cannot directly be applied to gaming applications. Work that focuses on application-specific power management for games was discussed in detail and weaknesses of the different approaches have been pointed out: Developed approaches are either too coarse to guarantee a good gaming experience and power savings

at the same time or are highly optimized for a particular game. Due to the huge number of games being published on a daily basis, however, such game-specific approaches that require the game's source code are not suitable. How these drawbacks can be overcome will be presented in the following chapter. Besides work that is related to CPU power management, we in addition gave an overview on work that has been performed to reduce the power consumption of peripherals like the display, GPU and memory. All of this work discussed is orthogonal to the techniques that are presented in the remainder of this thesis and can be combined with our methods to provide an overall optimized system power consumption.

3

Game workload prediction

Accurate workload prediction is the key to efficient dynamic voltage and frequency scaling. In this chapter we will systematically analyze PID controller-based workload prediction for games which has been proposed in the past [52], identify weaknesses of the approach and present prediction techniques, like autoregressive model-based predictors, that are better suited to accurately predict the highly dynamic game workload.

3.1 Contributions and related work

Control-theoretic techniques have been successfully applied to predict the future workload of video applications [124,154] and games [51,52]. It was shown that proportional-integral-derivative (PID) controllers can be leveraged to predict the workload of future frames of a game, based on the timing of previously processed frames and past prediction errors. However, for both, video applications and games the PID *gain values* had to be hand-tuned. In other words, the *proportional*, *integral* and *derivative* gain values had to be carefully chosen in order to maximize both power savings and the quality of the game play (measured by the percentage of frames missing the deadline). Some important questions were left open by this line of work.

First, if the PID gain values are tuned on the basis of one game application or a selection of game plays, then how robust or sensitive is the resulting controller to new game

applications or to a different selection of game plays in the same application? Here, a *game play* refers to particular sequences of scenes in the game or inputs provided by the user. Second, the PID controller-based prediction was evaluated on a relatively old game Quake II from id Software (because it is an open-source game), and in particular with it set to the *software rendering* mode. In other words, it was assumed that the mobile device did not have a graphics processing unit (GPU) and all the graphics processing had to be done in software on the CPU. Since the introduction of Quake II in 1997, the workload characteristics of games underwent substantial changes, GPUs are now available on mobile devices and the CPU increasingly processes complex physics or AI related tasks. Hence, how does the PID-based prediction scheme work for more modern games with higher workload variation and less inter-frame correlation? Finally, is it possible to design workload prediction schemes that do not require game-specific manual tuning of parameters, so that they work on game plays or even games that are not a priori known?

The contributions of the work presented in this chapter are as follows:

- We study the influence of PID gain values on the quality of game play (i.e., the number of frame deadline misses) and the achieved power savings for Quake II as well as for more recent games with hardware-based graphics rendering. Our results reveal that if PID gain values are not individually tuned for each game play, the controller might become unstable, resulting in a significant performance degradation. This shows that such PID-based prediction mechanisms, while extensively studied in the power management literature, cannot be practically applied to games, especially when the game play or game is not a priori known.
- In order to avoid this hand-tuning of PID gain values, we propose a self-tuning least mean squares (LMS) linear predictor. It achieves power savings and frame deadline misses that are comparable to those from a carefully hand-tuned PID controller, while the parameters of the LMS linear predictor do not necessarily need to be tuned for each play individually to provide an overall good performance.
- When we evaluated the LMS linear predictor on a set of recent games, we observed higher workload variations and reduced inter-frame correlation for games utilizing hardware-accelerated rendering. This diminished correlation affects the self-tuning process of the LMS linear predictor, which now might become unstable and provide inaccurate prediction results. To solve this problem, we next study autoregressive moving average (ARMA) models for workload prediction. Our results show that an ARMA model, that is tuned offline, performs well for a variety of games with both software-based and hardware-assisted rendering. Runtime power management techniques based on such prediction models show significant power savings, with very little perceived deterioration in the game quality.

We introduce a Microsoft *DirectX*-based DVFS framework to investigate different power management policies for more recent games for which, unlike Quake II, the source code is not available. Hence, they cannot be instrumented to record the processing times of individual frames. In this chapter, we leverage a lightweight dynamic-link library (DLL)

injection [103] technique to acquire required timing informations of closed-source DirectX games, while techniques that can be applied to Android-based systems will be described in detail in the next chapter.

Related work: PID controllers are commonly used in industry for system control [10, 161]. In the context of mobile computing PID controllers have been successfully applied to dynamic thermal management (DTM) [41, 136]. DTM aims to limit a processor's temperature by throttling or scaling the processing frequency to, e.g., avoid emergency shutdowns. It is shown that the CPU's temperature can be effectively stabilized using PID controllers. In [94], the processor's speed is adapted using a PID controller to regulate the fill-level of a MPEG decoder's play-out buffer. Marchesan et al. [98] leverage PID controllers to stabilize the task throughput on a multi-processor system-on-chip (MPSoC) by scaling the frequency of each core individually. In [150], the workload prediction required for DVFS is treated as a classical control theoretic problem, where the future load on a processor is predicted based on a discrete PID controller. The load estimate is then used to scale the processing frequency. It is shown that choosing the PID controller's weights is straight forward due to its low sensitivity to gain variations. The approach outperforms history-based approaches like simple averaging for benchmarks taken from the Media-Bench suite [84]. In [51, 52] a discrete PID-based predictor is used to forecast the workload of game frames. As discussed above, the PID gains were tuned manually and only for one game play of Quake II. In Section 3.4 we will show that in the case of gaming workloads, the PID controller is highly sensitive to gain variations. This complicates choosing correct gain values and makes the PID controller, despite its simple and well-studied nature, unattractive for game workload prediction.
In the work of Sinha et al. [134,135] significant power savings were obtained using a LMS-based workload prediction for DVFS. Akyol et al. [5] estimate the future complexity of video decoding tasks based on normalized LMS (NLMS) linear predictors. Similar to PID controllers, LMS linear predictors have as well been studied in the context of dynamic thermal management [19].
Coskun et al. [29] use an ARMA model to predict core temperatures on a MPSoC. The forecast allows a proactive reaction by migrating tasks to lower the utilization of individual cores. Liu et al. [91] interpret the sampled utilization of hard discs as time series and test different autoregressive (AR) and moving average (MA) models to predict the future utilization. It is shown that already an AR(1), i.e., a model of order $n = 1$, yields significant power savings with a minimal prediction overhead. In [30] an ARMA(1,1) model is successfully used to model the network traffic of Quake IV.

In order to predict the future workload, for example using ARMA-based predictors, timing information is required from the game which is commonly not available since most modern games are closed-source. To still get insight into modern closed source games, we leverage a technique called DLL-injection [103]. When applied to the DirectX library used by games, DLL-injection allows intercepting all graphics calls issued by the game (see Section 3.3.2). In the gaming context, DLL-injection has been mostly used for cheating: A famous cheat called wall-hacking [61,83] changes rendering commands of the game such that wall textures, e.g., the brick structures, are replaced by simple color schemes

while enemies are brightly colored. Thereby, the contrast is dramatically increased and players can aim easier and quicker. To the best of our knowledge, this is the first time that DLL-injection is leveraged for game power management.

Organization of the chapter: In the next section we briefly outline the main features of DVFS schemes for games. This is followed by a description of software- and the hardware-rendering schemes in games. Our simulation setup for tuning PID gain values is presented in Section 3.3.3. Next, the PID-based power management scheme is described in Section 3.4. Results using the PID controller for both software- and hardware-based rendering are then discussed. Next, we introduce the LMS linear predictor (see Section 3.5) and present the evaluation results in Section 3.5.2. We show the improvements over the PID controller, together with its limitations when used for games that leverage hardware acceleration for rendering. In Section 3.6 we discuss how time series analysis may be useful for modeling gaming workloads. Our results obtained from time series modeling, in particular from AR and ARMA models are then presented. Finally, in Section 3.7 power measurements are outlined before concluding with a summary.

3.2 DVFS for game applications

DVFS schemes for games primarily depend on estimating future game frame workloads. We evaluate several workload prediction techniques in terms of their performance and suitability. All the discussed techniques consist of two parts, an offline and an online phase.

Each predictor has a set of parameters, e.g., the predictor gains and the prediction order, which need to be determined during the **offline phase** before the predictor can provide a good performance in the online phase. Towards this, we record a sample game play based on which we tune the predictor's parameters. Such a sample game play can only resemble a small part of the game since not all future game plays of an interactive game application can be a priori known. Hence, the robustness of the predictor, i.e., its sensitivity to changes in workload characteristics, is of great importance. The ideal predictor should be tuned only once and then provide good performance for all future game plays. As the parameter tuning is done offline, it is not time critical.

The **online phase** of the DVFS scheme has the following structure (as described in Chapter 1): (i) The workload of the $i+1$-th frame is predicted from the workloads c and estimation errors e of previous frames:

$$\tilde{c}[i+1] = predictor(c, e)$$

(ii) based on prediction results and the desired frame rate, the required clock frequency of the processor is computed, (iii) the processor's frequency is scaled accordingly, and (iv) the frame is eventually processed (involving game AI, physics-related computations,

etc.) and rendered, and finally (v) based on cycle-accurate measurements of the workload $c[i+1]$ the estimation error $e[i+1]$ is computed

$$e[i+1] = \tilde{c}[i+1] - c[i+1]$$

and fed back to the workload predictor. Since the online phase is time critical, the overhead of the different steps will be discussed in Section 3.7.1.

DVFS performance metrics: We introduce two metrics to evaluate the prediction quality and the performance of the resulting DVFS scheme. The first metric is the average power $\overline{P}$ consumed during a game play. The power consumption depends on the processing frequencies f_i chosen from the set of available frequencies $\mathcal{F}$. $\overline{P}$ is measured as described in Section 3.7 or estimated as described in Section 3.3.3.

The second metric is the percentage of frames missing their deadlines, d. If the i-th frame's processing time $t[i]$ is greater than the deadline $d = 1/fps_{desired}$ the frame is said to have missed its deadline, where $fps_{desired}$ is the desired frame rate. Note that unlike in video processing applications, a frame that misses its deadline is not dropped; it only leads to a possibly poor gaming experience. Further, there is a clear dependency between the two metrics: Always using the smallest available processing frequency f_{min} will minimize $\overline{P}$, but will lead to the maximum number of frames missing their deadline (d_{max}). Using the largest available processing frequency instead will lead to a high power consumption, but will result in the minimum number of frames missing their deadline (d_{min}). An optimal predictor (using an oracle) would allow optimizing d, $\overline{P}$ or a combination of both.

Choosing a target frame rate: An important decision is the choice of the target frame rate $fps_{desired}$ as it influences both, the percentage of frames missing their deadline d and the average power consumption $\overline{P}$. Lowering $fps_{desired}$ will result in a lower workload, and therefore it is more likely that the frame can be processed in time with a lower frequency. This in turn will lead to higher power savings, but might lead to poor gaming experience. User perception studies reported by Claypool *et al.* [28] show that the game frame rate has a high impact on the perceived game quality. The perception varies from game to game, i.e., a strategy game's frames per second (fps) demand is likely to be lower than the desired fps in case of a fast first person shooter game. For this work we have chosen the target frame rate for each game such that the perceived game quality appeared to be optimal for us, e.g., for Quake II this was a frame rate of 30. This implies that each frame has a deadline of 1/30-th of a second.

3.3 Architectural setup

The correct choice of a prediction technique highly depends on the underlying architectural setup. Therefore, we will first describe the two hardware setups used in this chapter, before we introduce the different prediction techniques. Note that in this chapter we use desktop game applications and setups to allow comparability to previous work [54]. In

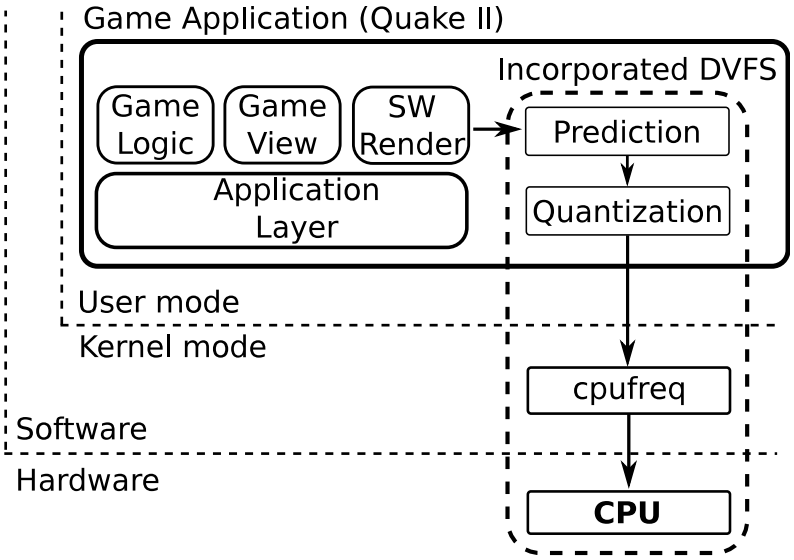


Figure 3.1: Experimental setup for software-based rendering DVFS

Chapter 4 it is shown how the implementations presented in the following can be ported to Android-based mobile platforms. Further, it is verified that all results presented in this chapter for desktop games as well hold true for mobile games.

The first setup consists of a software-based rendering setup similar to the one used in [54]. Low-cost mobile devices without hardware accelerated rendering will fall under this category. The second setup uses a graphics processing unit (GPU) to render the game's graphics. This setup is now typically found in most high-end mobile devices like smartphones and tablet PCs.

3.3.1 Software-based rendering setup

We employed the Quake II game engine which forms the core of some of the most popular first person shooter games like Raven Software's Soldier of Fortune, Anachronox from EIDOS and Activision's Heretic II. Quake II was chosen for two reasons: i) We could compare our results to other related work that studied Quake II [6, 50, 54]. ii) Quake II uses software-based rendering and its engine forms the core of a variety of other games as mentioned above. The source code of Quake II is available under GNU public license and has been modified to incorporate the proposed DVFS power management algorithms (see Figure 3.1). This modification can be applied to any open-source game.

The software video mode of the game was set to mode 5 which corresponds to a frame resolution of 960×720 pixels. The desired frame rate was set to 30 fps for all Quake II related experiments. The modified source code was compiled in release mode with processor specific optimizations and the game plays were run on Ubuntu 9.10 operating system with Linux kernel version 2.6.31-20-generic.

Table 3.1: Workload statistics for the used Quake II game plays

Quake II Game Play	Avg. workload $\overline{C}$ $\left[\frac{cycles}{frame}\right]$	Deviation σ $\left[\frac{cycles}{frame}\right]$	Target FPS	Frames missing their deadline % d_{max}	d_{min}
Explore-1	3.7e+07	3.7e+06	30	0.7	0.0
Explore-2	3.8e+07	3.2e+06	30	3.3	0.0
Shooting-1	4.1e+07	4.9e+06	30	71.8	0.0
Shooting-2	4.1e+07	4.1e+06	30	67.5	0.0
Level-2	4.0e+07	6.4e+06	30	66.6	0.2
Massive-1	4.5e+07	7.7e+06	30	86.5	1.8

The experiments were performed on a laptop equipped with a 1.86 GHz Intel® Pentium® M Processor and 1.5 GB RAM. This processor supports Enhanced Intel SpeedStep® Technology and offers frequency scaling between five different frequency levels that correspond to 800 MHz, 1066 MHz, 1333 MHz, 1600 MHz and 1866 MHz. In order to obtain a precise processor cycle count, the cycle measurements were performed with the help of the RDTSC (read-time stamp counter) instruction. The RDTSC instruction to log each frame's cycle count was incorporated into the source code of Quake II, along with the DVFS algorithms.

Selection of game plays: Quake II allows recording of game plays. These recordings include everything that is required (e.g., user input) to re-run exactly the same game play similar to a video, along with performing exactly the same computations. This allowed us to reproduce the measurements for different runs of the same game play under identical settings, but with different power management policies.

We recorded four short game plays among which two (i.e., Shooting-1 and Shooting-2) included highly dynamic scenarios involving events like enemy contact. The other two short game plays resembled an exploration phase of the game with comparatively low workload (i.e., Explore-1 and Explore-2). Additionally, we recorded a long game play (i.e., Level-2) with average workload and dynamics. The dynamic behavior of the predictor was also examined using Massive-1 which is a well-known Quake II benchmarking demo with relatively high CPU demand and workload variation. Several runs were recorded for each game play to take the variations caused by the underlying OS into account. The resulting statistics of all game plays are shown in Table 3.1, where the average workload $\overline{C}$ and its standard deviation σ are given in terms of processor cycles per frame. It is obvious that Massive-1 has the highest average workload and standard deviation caused by the highly dynamic nature of this game play. On the other hand, Explore-1 and Explore-2 show the lowest $\overline{C}$ and σ. The minimum (d_{min}) and the maximum (d_{max}) percentage of frames missing their deadlines were obtained by running the processor at the highest (1600 MHz) and the lowest (800 MHz) frequency respectively. A certain percentage of frames miss their deadline even when the processor runs at the highest frequency, because the processor cannot support the maximum possible workload generated.

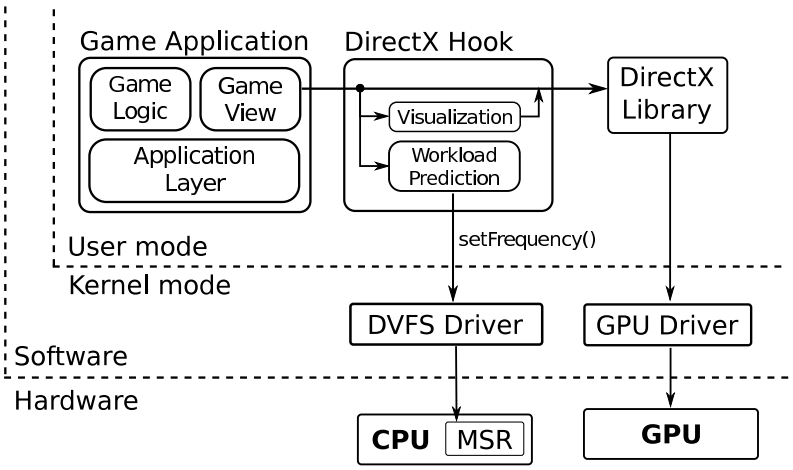


Figure 3.2: Experimental setup for hardware-based rendering DVFS

3.3.2 Hardware-based rendering setup

In the following, the hardware-based rendering setup is described, which is used to evaluate the prediction performance of games running on a HW-accelerated rendering system. Previously, rendering made up the major part of the total workload (in case of Quake II up to 90 percent). The advent of GPUs allowed game developers to offload most of the rendering workload from the CPU. Instead, the CPU is now increasingly used for complex artificial intelligence computations and realistic physics engines. Thereby, the workload characteristics of games have significantly changed.

The main advantage of the setup presented below over the previously presented approach is that (i) we can evaluate modern games that use hardware-accelerated rendering, and (ii) that we are no longer restricted to open-source games. The current implementation allows applying our DVFS algorithm to any DirectX 9 based application and can be easily adapted to support DirectX 10/11, OpenGL or OpenGL ES (see Chapter 4). To incorporate our power management into recent closed-source games, we utilized the interface between the application and Microsoft's DirectX rendering API. As shown in Figure 3.2, a game performs API calls to the Microsoft Direct3D run-time library to initiate the rendering of the scene. We intercepted all calls made by the game to the Direct3D run-time library. This was done using a technique called dynamic link library (DLL) injection provided by the Microsoft Detours Library [103]. When the game starts up, it loads the Direct3D library. The DLL injection inhibits the loading of the original DLL and instead forces the game to load our own library into its context. Consequently, instead of the original Direct3D functions, our code is executed. This enabled us to profile the game, to run our workload prediction and DVFS scheme and to render a visualization on top of the game (see Figure 3.3).

A game signals the beginning of a new frame to the GPU with the DirectX command `BeginScene`. At the end of the frame, the `EndScene` together with the `Present` command is called, thereby informing the GPU that the frame can be shown. The interarrival time

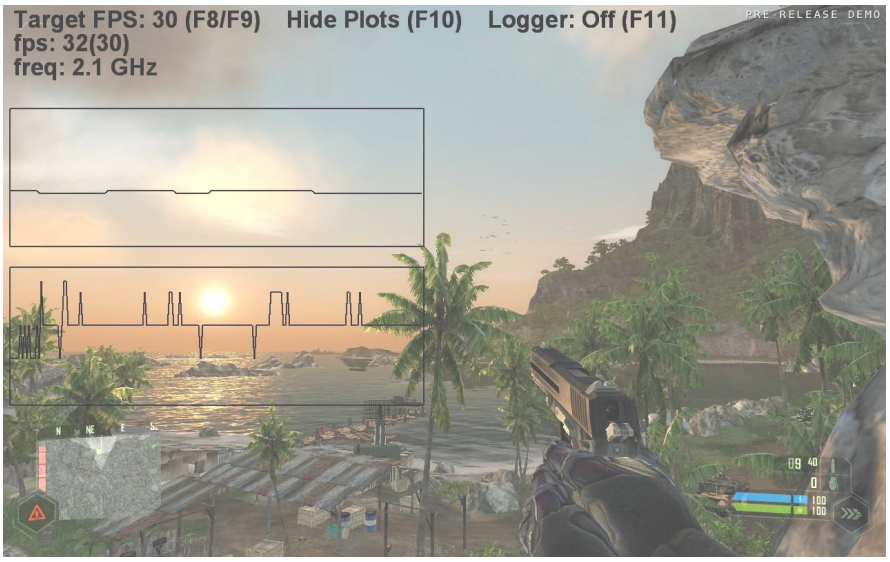


Figure 3.3: Screenshot from Crysis with integrated visualization of frame rate (top left) and processor's frequency (middle left)

between `Present` commands gives us the frame execution time. Based on these execution times the predictor estimates the workload of the next frame and computes the required processor's frequency. To actually scale the frequency of the processor, the processor's model specific registers (MSRs) have to be set accordingly. As this is only allowed in kernel mode, a driver is loaded at creation time of the DirectX device which allows setting the MSRs from user space (see Figure 3.2). Additionally, this interception-based approach allowed us to add an online visualization of e.g., the power savings and a user interface, which enables the user to influence the DVFS algorithm and modify parameters like the desired frame rate.

For the experiments we used a desktop PC with an Intel® Core™ 2 Quad QX6700 with 2.66 GHz, 2048 MB of RAM and a NVIDIA Geforce 8800 FX graphics card and running Microsoft Windows XP. The processor supports five different frequency levels that correspond to 1.6 GHz, 1.86 GHz, 2.1 GHz, 2.4 GHz and 2.66 GHz. We used the Windows system API functions `QueryPerformanceCounter()` and `QueryPerformanceFrequency()` to measure the frame execution times. In this work the frequency and voltage of the four cores was always scaled equally as the focus of this chapter is on the evaluation of suitable workload predictors for game applications.

Selection of games: We have chosen three popular games, each from a different genre, to evaluate the performance of our power management scheme: A first person shooter named Crysis from Crytek, a racing game named Need for Speed - Shift (NFS) from Electronic Arts and Ubisoft's strategy game World in Conflict (WIC). The average workload $\overline{C}$, the workload's standard deviation σ, d_{min} and d_{max} are given in Table 3.2 for six recorded game plays (two for each game). The workloads' standard deviation for games that use hardware rendering are significantly higher compared to the one's in Quake II. The

Table 3.2: Workload statistics for the used DirectX game plays

DirectX Game Play	Avg. workload $\overline{C}$ $\left[\frac{cycles}{frame}\right]$	Deviation σ $\left[\frac{cycles}{frame}\right]$	Target FPS	Frames missing their deadline % d_{max}	d_{min}
NFS-1	5.5e+07	1.0e+07	30	50.22	0.67
NFS-2	6.0e+07	1.1e+07	30	68.83	1.56
Crysis-1	3.8e+07	1.4e+07	40	33.92	0.66
Crysis-2	4.6e+07	3.7e+07	40	70.51	1.89
WIC-1	3.3e+07	8.0e+06	40	15.78	0.10
WIC-2	4.0e+07	8.4e+06	40	65.20	0.15

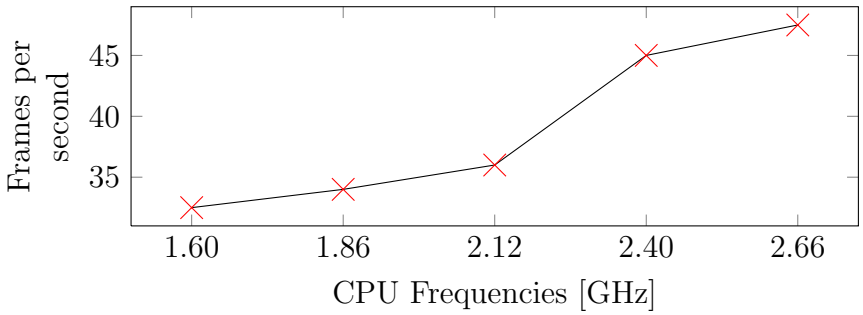


Figure 3.4: Average frame rate under varying CPU speed in Crysis

major difference between the two architectural setups is the composition of the workloads. A game's workload is in general composed of rendering, AI, physics and game logic. Computations like AI or physics are not necessarily frame-based and therefore can change more abruptly if measured in cycles per frame. The rendering workload on the other hand is of course frame-based, and therefore changes are likely to be slower. In case of Quake II, up to 90 percent of the CPU's time is spent for rendering. For games with hardware-rendering the CPU time is instead used for complex AI or physics computations. As these computations are not necessarily done on a frame basis, the workload no longer changes in a continuous manner. This explains the observed difference between the workload's standard deviations, and this will later influence the choice of appropriate predictors.
This work targets power management schemes for the CPU. If a game is mainly GPU-bound for the available CPU frequencies, then it could always be run with the lowest available frequency and no DVFS or workload prediction would be required. To present meaningful results, we have chosen realistic graphics settings for each game and verified that the games are not always GPU-bound. Figure 3.4 shows the resulting average frame rate of Crysis for the available processor speeds. It can clearly be seen that the game is not GPU-bound for the chosen settings (Resolution 1024×768, no Anti-Aliasing, all detail levels set to medium). Note that there still might be scenes for which a game becomes GPU-bound in-between. Here, it would be beneficial to come up with an additional power management scheme for the GPU.

Table 3.3: Run-times for varying number of simulation runs

No. of Simulation Runs	t_{sim} [s]	t_{game} [s]	Speedup
400	116.4	78.7e+03	676
1600	206.3	31.2e+04	1515
442401	15650	86.3e+06	5512

3.3.3 Simulation setup

The highly dynamic nature of game workloads and additional variations introduced by the underlying OS necessitate an exhaustive exploration of the space of workload predictor parameters, as will be elaborated later in this chapter. Towards this, we developed a simulation environment for a systematic evaluation of the performance of the PID controller, the LMS and the ARMA-based workload predictor.

The algorithm for DVFS (see Section 3.2) is replicated in the simulation, where the processing of frames and the workload measurement is replaced by a workload *model*. This workload model is based on recorded workload profiles.

In contrast to video applications, in games the content of every frame and its workload depends on the user action and the processor frequency that was used to render the past frames: Let the i^{th} frame at time $t[i]$ require $c[i]$ clock cycles and $f_j[i] \in \mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be the corresponding processor frequency used to render the i^{th} frame. The time $\Delta t[i]$ taken for rendering the i^{th} frame is then approximated by $c[i]/f_j[i]$. Further, the next frame will be rendered at time $t[i+1] = t[i] + \Delta t[i]$. After the i^{th} frame has been rendered, the physics engine calculates the player's new position based on the player's speed and $\Delta t[i]$ (which is the real passage of time). The position of the player and the next frame's content therefore depends on $\Delta t[i]$ and hence on the frequency $f_j[i]$ (when the frequency is higher, more frames are used to "fill" a certain time interval).

This dependency prohibits the direct usage of recorded workload profiles. To get around this, we assume "linear" behavior of the workload profiles. Our experiments showed that this assumption is valid over small time scales, as considered in our case. Thus, for each available processor frequency $f_j \in \mathcal{F}$, the corresponding workload profile C_{f_j} is recorded and transformed from the *frame number* to the *time* domain by interpolating the missing values. For each frame that is *pseudo*-processed in simulation with frequency f_j, the corresponding workload profile C_{f_j} is evaluated.

This workload model together with the replicated steps of the DVFS scheme now allows accurate approximation of the system behavior and an evaluation of different controller settings for DVFS in terms of the performance metrics (see Section 3.2). The runtimes of the simulation compared to concrete runtimes of the games are given for different runs in Table 3.3. A speedup of 5512× is achieved with a Mathworks MATLAB implementation, which clearly shows the advantage of using a simulation-based approach for tuning the controller parameters (gain values) of our workload predictors, as will be explained later.

3.4 PID controller-based prediction

Now that we have explained our experimental setup, in what follows we describe various workload predictors and their evaluations. We start with the PID controller-based workload prediction which has been successfully applied to various power management problems. In the following, we are going to describe the theoretical background and challenges of using PID controllers for game workload prediction.

The input signal computed by a PID controller consists of three components,

1. *Proportional:* $P_{comp}(t) = K_P \cdot e(t)$
2. *Integral:* $I_{comp}(t) = \frac{1}{K_I} \cdot \sum_{T_I} e(t)$
3. *Derivative:* $D_{comp}(t) = K_D \cdot \frac{e(t) - e(t-T_D)}{T_D}$

where $e(t)$ is the error signal, K_P, K_I, K_D are proportional, integral, derivative gains respectively. T_I and T_D are intervals for integral and derivative components respectively.

The output of the PID controller is given by

$$PID_{output}(t) = P_{comp}(t) + I_{comp}(t) + D_{comp}(t).$$

Let $c[i]$ and $\tilde{c}[i]$ be the respective actual and estimated workload values for the i^{th} frame in terms of clock cycles. Here, the goal is to predict the workload $\tilde{c}[i+1]$ of the $(i+1)^{th}$ frame by utilizing the actual workload $c[i]$ of the i^{th} frame and the PID control signal $PID_{output}(t)$, i.e., $\tilde{c}[i+1] = c[i] + PID_{output}[i]$. Towards this, we compute $PID_{output}[i]$ with $e[i] = c[i] - \tilde{c}[i]$ being the error signal and

$$\begin{aligned} PID_{output}[i] &= K_P \cdot e[i] \\ &+ \frac{1}{K_I} \cdot \sum_{j=0}^{n} e[i-j] \\ &+ K_D \cdot \frac{e[i] - e[i-1]}{\Delta t[i-1]} \end{aligned}$$

3.4.1 PID controller's stability

The choice of PID controller gains K_P, K_I and K_D is crucial for the performance of the predictor. The predictor directly influences the choice of the processor's frequency. Ideally, it will choose a frequency sufficient to complete the frame just in time and therefore with the lowest possible power consumption. If the predictor's parameters are chosen

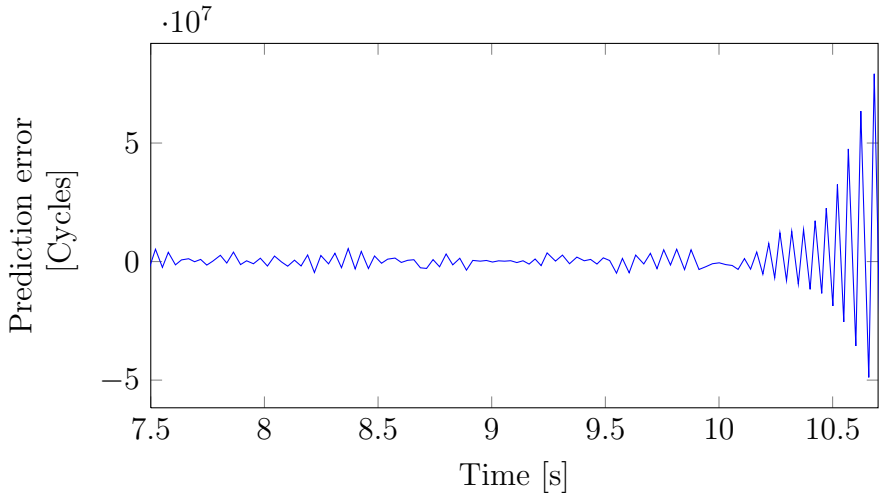


Figure 3.5: Prediction error in case of an unstable predictor

incorrectly the predictor may become unstable. A typical plot of a prediction using an unstable controller is shown in Figure 3.5. The prediction error and hence the predicted workload starts to oscillate and becomes infinitely large over time. In terms of processor frequency settings, this results in a periodic switching between highest and lowest frequency. This of course is highly inefficient in terms of power consumption and will also lead to a considerable loss in game quality.

Figure 3.6 shows the distribution of stable controller gains for a particular game play (i.e., Shooting-2). It may be noted that only a small portion of the entire space of the controller gains ensures the predictor's stability. The controller gains with reasonable prediction quality (i.e., smaller than 10 % frame deadline miss) are indicated by the points. Note that the number of such points is limited and distributed over the entire space of stable controller gains. Hence, identifying gain values that lead to a controller with acceptable performance is a non-trivial task. In the following subsection we show how suitable gain values may be chosen.

3.4.2 PID performance space

As mentioned in Section 3.2, the performance of the PID controller and all other workload predictors is quantified using two metrics, the percentage of frames missing their deadline and the average power consumption. We obtain a performance space by plotting the values of the metrics corresponding to various sets of PID controller gains. For example, Figure 3.7 shows the resulting performance space of the PID controller for the Quake II game play Level-2. Every point corresponds to results obtained with a specific choice of PID controller gains. The Pareto-front marks the optimal choice of points. These are the points of interest and are found by an exhaustive simulation. It may be noted that the average power consumption, here given for the laptop, is in the range of 22 to 23 Watts resulting in maximum possible savings of 35 % (compared to the maximum power con-

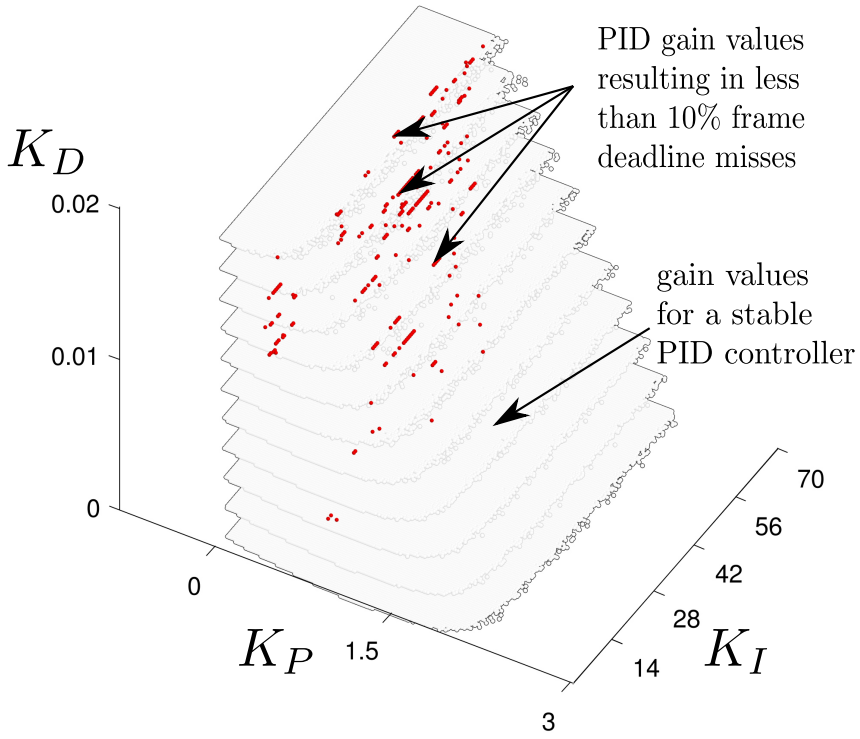


Figure 3.6: PID controller gains in the range of interest for Shooting-2. Each plane indicates a stable choice of PID gains (with $K_D = constant$ and K_I, K_P varied)

sumption of the laptop, which is 34 Watts). However, reducing the power consumption to 22 Watts comes at the cost of an unreasonably high number of frames missing their deadline (i.e., over 25 %). Moreover, the variation in power consumption is small compared to the maximum average power consumption in a laptop. The same behavior has been observed for the desktop PC. It may also be observed that the percentage of frames missing their deadlines is highly influenced by the choice of the gain values. Hence, we chose the gain values with the lowest percentage of frame deadline misses (to maximize game quality). Clearly, a systematic identification of suitable controller gains is necessary for each game play.

Software-based rendering: To investigate the influence of workload variations, we ran exhaustive simulations for the Quake II game plays listed in Table 3.2. For each game play, we used our simulation setup to explore the effect of the controller gain values. Such exhaustive search results in performance spaces similar to the one shown in Figure 3.7. For each game play we selected the set of gains resulting in the lowest rate of frame deadline misses, i.e., $\mathbf{set_{Explore-1}}(\times)$ is the optimal set for game play Explore-1 (see Figure 3.8).

In a complex game it is very unlikely that a player generates the same workload twice. Hence, an important issue is the predictor's *robustness* to changes in the workload characteristics. To evaluate this robustness we used the PID gains, which were optimized for one game play, for the workload prediction of the remaining game plays. As shown in

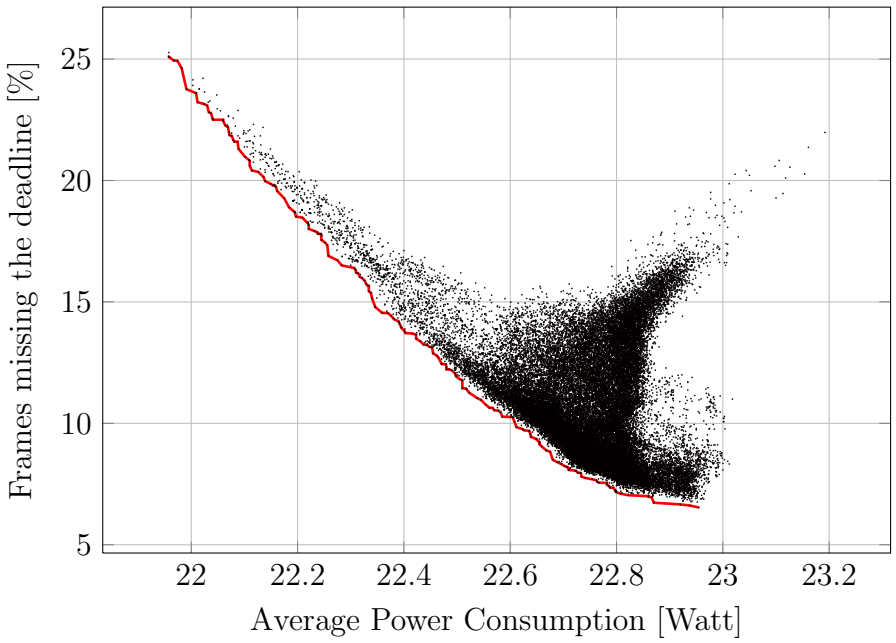


Figure 3.7: Performance space for different choices of PID controller gains for the Quake II gameplay Level-2

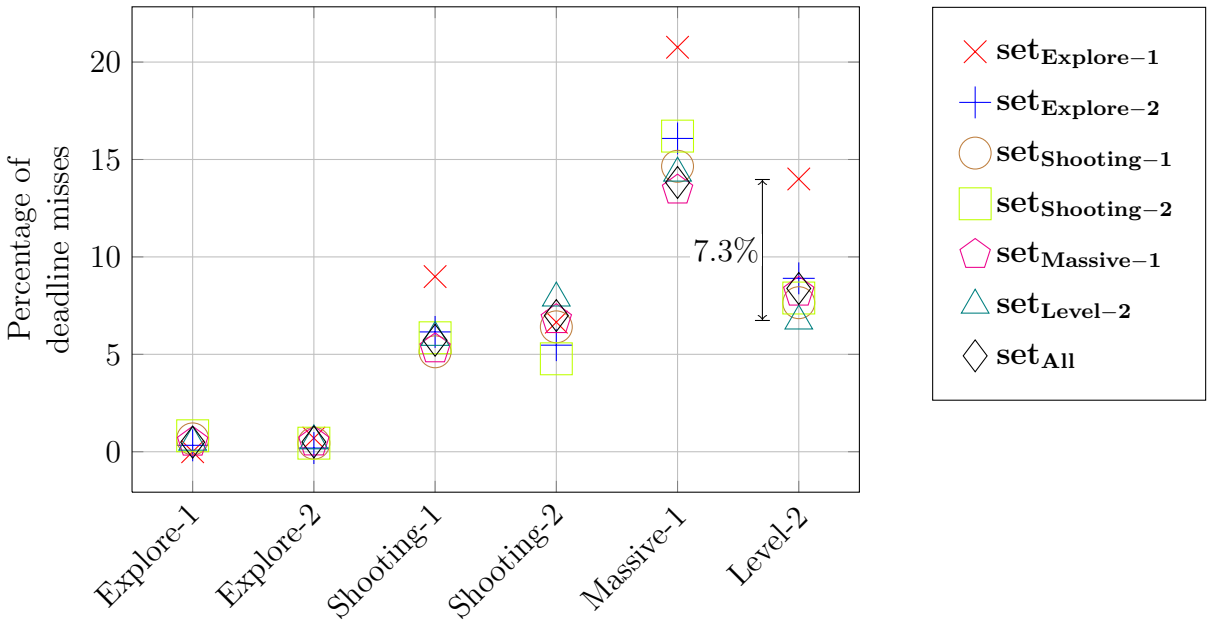


Figure 3.8: Performance evaluation of game-play-optimized PID gain sets together with the gain set tuned for all game plays (with software rendering setup)

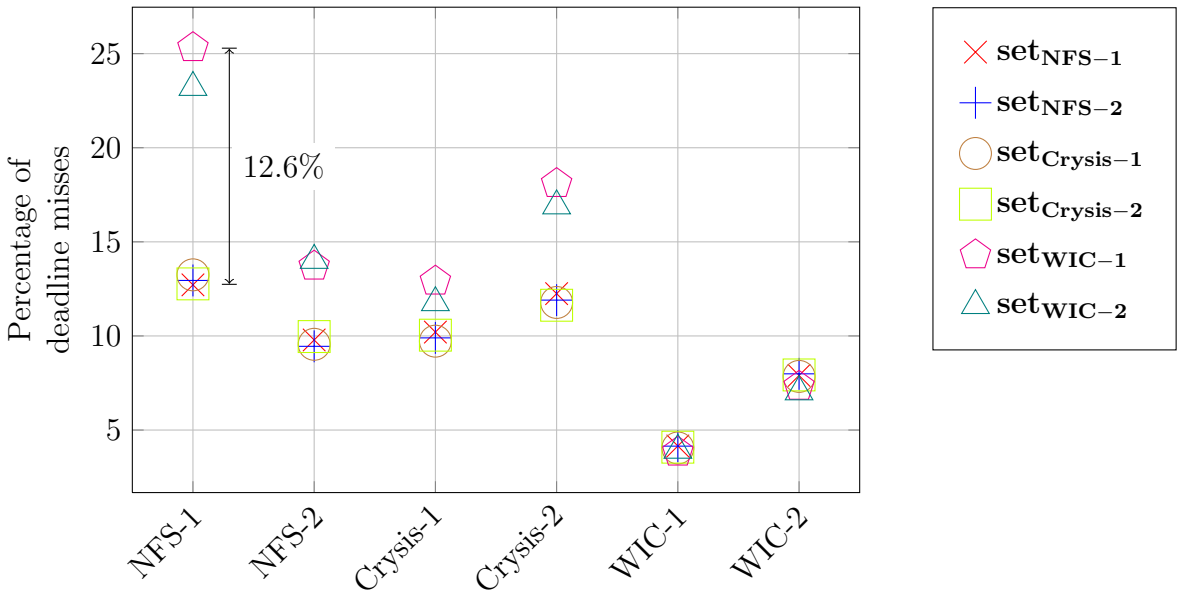


Figure 3.9: Performance evaluation of game-play-optimized PID gain sets (with hardware rendering setup)

Figure 3.8, we observed inferior performance if we used gain values that were optimized for one game play for other game plays. For example, using $\mathbf{set_{Explore-1}}$(×) for Level-2 will increase the percentage of frame deadline misses by 7.3 %.

The PID gains can also be optimized with all the game plays taken together, i.e., $\mathbf{set_{ALL}}$(◇) in Figure 3.8. This set has been computed by merging the performance spaces of all game plays. Nevertheless, the controller gains again need re-optimization in case a new game play is considered. For example, we optimized the PID gains by considering all the game plays listed in Table 3.2 except for the game play Level-2. These PID gains were then used for Level-2, which resulted in an unstable predictor for Level-2.

Hardware-based rendering: The same measurements were also repeated for hardware rendering. Unfortunately, the option to record game plays like in Quake II is not offered by any recent games that use hardware rendering. To address this issue of reproducibility, the predictor's performance was simulated, first using very long pre-recorded workloads. Once a good setting was found the gains were evaluated online in the game and results were thereby verified.

Figure 3.9 shows the resulting percentages of frames missing their deadlines for the DirectX games listed in Table 3.2. The gains have been tuned individually for six game plays in total (two per game). A cross-validation was performed with the other game plays. As can be seen in the figure, the performance drops significantly when a set of PID gains, which has been tuned for one particular game play, is used for a different game or game play, e.g., using $\mathbf{set_{WIC-1}}$(⬠) for NFS-1 increased the number of frame drops by 12.6 %. Further, for all games the controller became unstable after a particular number of frames.

Based on the measurement results from both, the software- and hardware-rendering setup, we conclude that the PID-based predictor's performance depends on the nature of the game play and hence the controller gains should be optimized for each game play individually. Since all game plays cannot be known in advance (because they depend on the player), we conclude that PID-based prediction, which was successfully applied to video power management, is not suitable for game applications in real-life settings. Instead, a *robust* predictor is required, that needs to be calibrated only once, and that thereafter provides good performance for a priori unknown game plays. These outcomes motivate the use of a *self-tuning* algorithm.

3.5 LMS linear predictor

The least mean squares (LMS) linear predictor [58] is a statistical approach mainly used for parameter identification of various dynamical systems. Such approaches are suitable for systems that are *linear-in-parameters* (LIP), i.e., the output of the system can be modeled as a linear combination of system inputs and (unknown) system parameters. The LMS linear predictor learns the system parameters by recursively updating its weights over several iterations.

We used it to estimate the workload $\tilde{c}[i+1]$ of the $(i+1)^{th}$ frame by utilizing the actual workloads of the previous frames. Towards this, we represented the predictor's output as a linear combination of known workloads of previous frames and unknown predictor coefficients. If $c[i]$ represents the workload value of the i^{th} frame, then according to the general structure of a one-step LMS Linear Predictor, the predicted workload of the $(i+1)^{th}$ frame is given by

$$\tilde{c}[i+1] = \sum_{k=0}^{n-1} w_k[i]\ c[i-k] = W[i]^T C[i], \tag{3.1}$$

where n is the predictor's order, w_k, for $k = 0, \ldots, n-1$, are unknown predictor coefficients and

$$W[i] \;=\; [w_0[i], w_1[i], \ldots, w_{n-1}[i]]^T$$

$$C[i] \;=\; [c[i], c[i-1], \ldots, c[i-n+1]]^T$$

The goal is to learn $W[i]$ adaptively such that $\tilde{c}[i+1]$ in Equation (3.1) results in the minimum error $e[i] = \tilde{c}[i+1] \;-\; c[i+1]$. Therefore, from Equation (3.1) we have, $e[i] = c[i+1] - W[i]^T C[i]$. The unknown predictor coefficients are initialized to 0 and after each prediction step, the coefficients are updated according to Equation (3.2)

$$W[i+1] = W[i] \;+\; \mu \cdot e[i] \cdot C[i], \tag{3.2}$$

where μ is the *learning rate*.

In order to reduce the sensitivity of the learning process to the choice of the learning rate μ, we used a normalized LMS (NLMS) predictor that is given by

$$W[i+1] = W[i] \; + \; \frac{\mu \cdot e[i]}{||C[i]||^2} \cdot C[i], \tag{3.3}$$

with μ being between 0 and 2.

For an accurate modeling of the system and prediction of the workload, the coefficients $W[i]$ should converge after a sufficient number of iterations. However, this convergence is not guaranteed.

3.5.1 LMS weight convergence

The LMS Linear Predictor models the system accurately if the coefficients $W[i]$ converge to the statistical mean after a sufficient number of iterations. In the following we derive a bound which should not be violated repeatedly if convergence is desired.

According to Equation (3.1) and (3.3) the individual weights get updated by

$$w_k[i+1] = w_k[i] \; + \; \frac{\mu \cdot e[i]}{||C[i]||^2} \cdot c[i-k].$$

The above can be reshaped as follows,

$$w_k[i+1] = w_k[i] \; + \; (\alpha - 1) w_k[i] = \alpha w_k[i].$$

α is a scalar given by,

$$w_k[i](\alpha - 1) = \frac{\mu \cdot e[i]}{||C[i]||^2} \cdot c[i-k]$$

$$\Rightarrow \alpha = 1 + \frac{\mu \cdot c[i-k]}{||C[i]||^2} \cdot \frac{e[i]}{w_k[i]}$$

Clearly, for stable convergence of weights, we need $|\alpha| \leq 1$ which further implies

$$-2 \leq \tfrac{\mu \cdot c[i-k]}{||C[i]||^2} \cdot \tfrac{e[i]}{w_k[i]} \leq 0$$

$$\Rightarrow \tfrac{-2||C[i]||^2}{\mu \cdot c[i-k]} \leq \tfrac{e[i]}{w_k[i]} \leq 0$$

$$\Rightarrow |e[i]| \leq \tfrac{2||C[i]||^2}{\mu \cdot c[i-k]} \cdot |w_k[i]| \, .$$

It is evident that there is an upper bound on the prediction error $|e[i]|$, given by

$$\frac{2||C[i]||^2}{\mu \cdot c[i-k]} \cdot |w_k[i]| \,. \tag{3.4}$$

If $e[i]$ repeatedly goes above this bound, the LMS weights might not converge. Not converging and therefore continuously growing weights will result in an unstable predictor and thus high prediction error. Convergence, on the other hand, implies accurate approximation of the system using a LMS linear predictor.

Table 3.4: Percentage of frames for which at least one of the weights did not satisfy the convergence boundary

Quake II		DirectX	
Game play	Percentage	Game play	Percentage
Explore-1	0.06	NFS-1	64.08
Explore-2	0.04	NFS-2	0.02
Level-2	13.12	Crysis-1	87.29
Massive-1	0.03	Crysis-2	77.54
Shooting-1	0.05	WIC-1	44.39
Shooting-2	0.06	WIC-2	0.01

We have evaluated this equation for all game plays of both, the soft- and hardware rendering based setup. The results are presented in Table 3.4. As can be seen, for the Quake II game plays the bound is only seldom violated whereas for nearly all DirectX game plays violations to a large extend have been observed. From this we conclude that the LMS linear predictor models the Quake II game play's workload behavior correctly, but is not a suitable choice for the highly dynamic hardware rendering based workloads.

3.5.2 Performance evaluation

Software-based rendering: The performance of LMS linear prediction is determined by two parameters, the prediction *order* n and the *learning rate* μ. The order n of the predictor indicates the number of workload values of the previous frames being utilized to model its output (see Equation (3.1)). If the order of the predictor is too low, e.g. $n = 1$, then its coefficients will not be able to accurately model the output and will not converge. This in turn results in a significant reduction of the predictor's performance (see Figure 3.10). However, an unnecessarily high order, i.e., $n \geq 11$, does not improve the performance any further. We experimentally found that an order of 10 guarantees a good performance for all Quake II game plays.

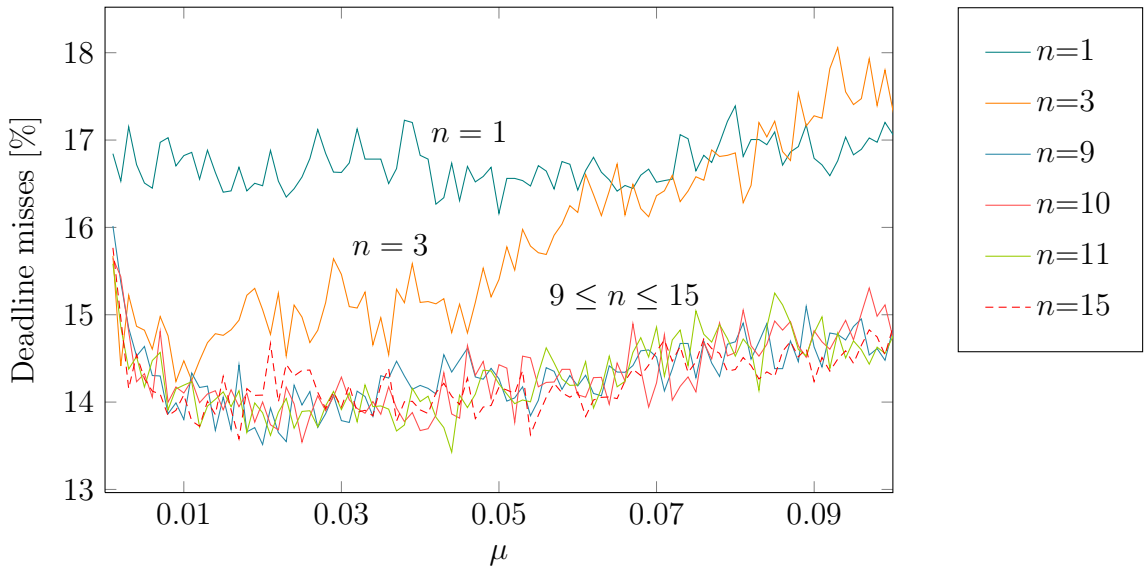


Figure 3.10: Impact of the order n and learning rate μ on the percentage of missed deadlines using LMS linear predictor for the game play Massive-1

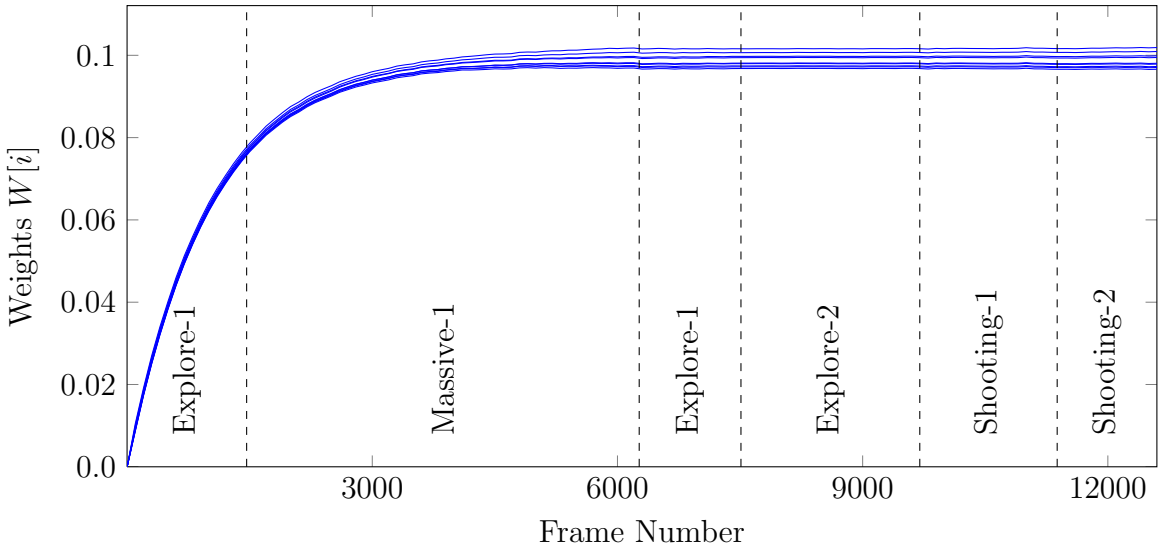


Figure 3.11: Convergence of LMS-weights $W[i]$ with varying game dynamics ($\mu = 0.074$, $n = 10$)

Using this order, the boundaries given by Equation (3.4) were seldom violated (at maximum by 13.12% for Level-2) thereby indicating a convergence of the weights. Figure 3.11 depicts the variations of the weights over a sequence of frames. It is clear that the weights converge after 6000 frames. As indicated in the figure, we initiated switches between Quake II game plays during the simulation to verify that convergence is preserved under changing system dynamics. Therefore, Quake II system dynamics can be accurately approximated by a LMS linear predictor.

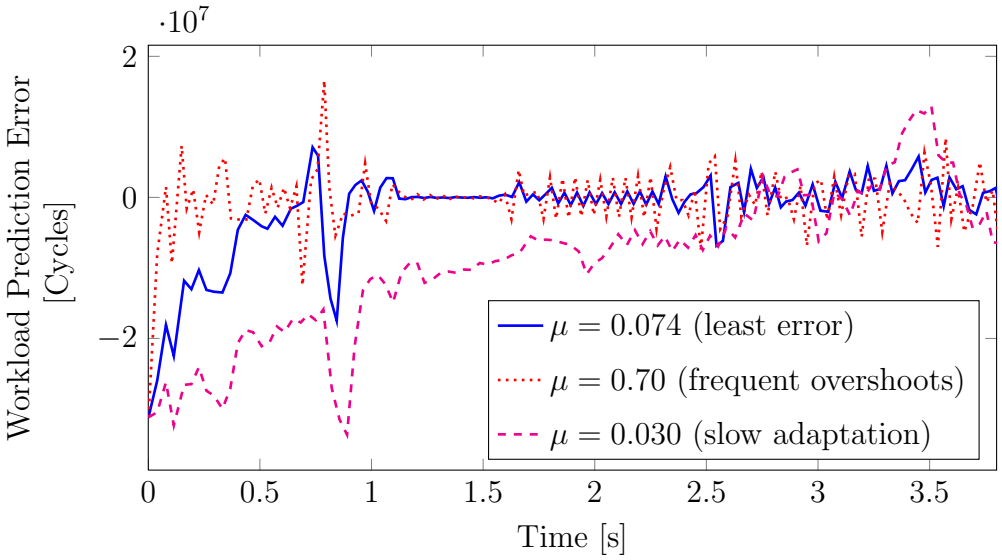


Figure 3.12: Comparison of resulting error using different learning rates μ for the Quake II gameplay Level-2

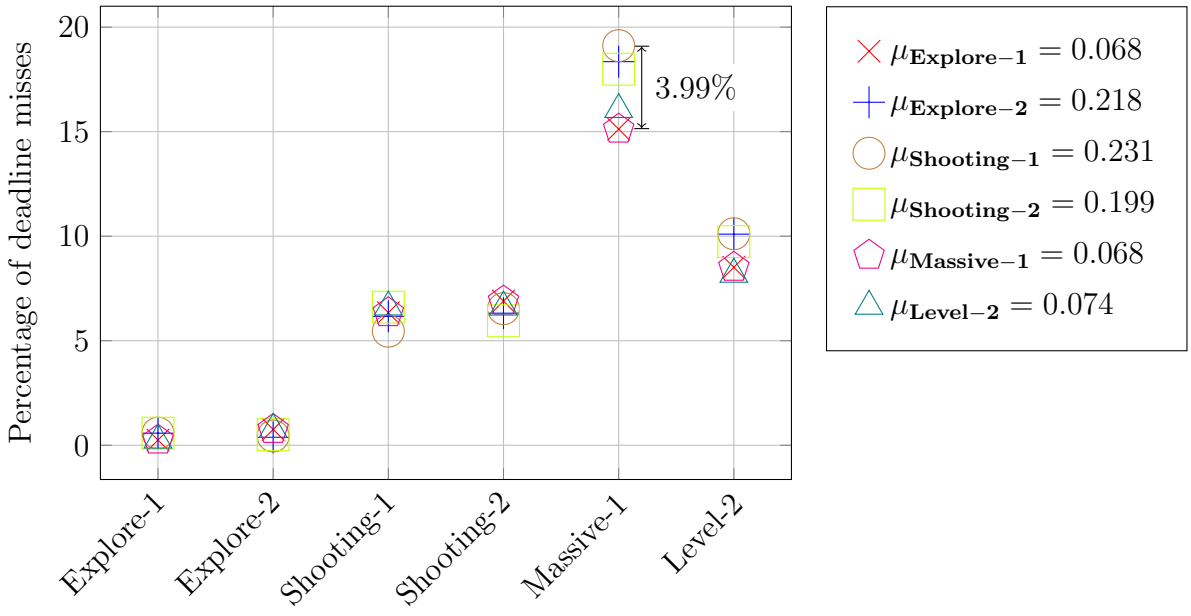


Figure 3.13: Performance evaluation of game-play-optimized LMS learning rates μ (with software rendering setup)

The second parameter that affects the quality of prediction is the learning rate μ. As seen in Figures 3.10 and 3.12, a very small learning rate μ results in a high prediction error as the learning process is too slow for appropriate adaptation of the weights. On the other hand, a high learning rate results in overdrive effects, i.e., the weights also learn noise. This especially affects dynamic scenes for which the resulting processor frequency varies abruptly.

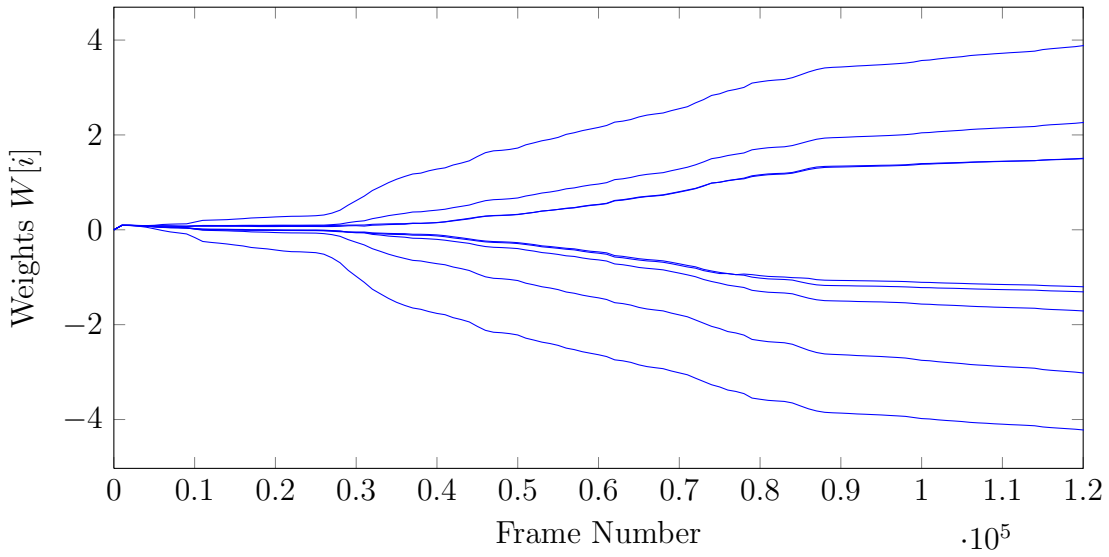


Figure 3.14: LMS-weights $W[i]$ over time for game play Crysis-1 ($\mu = 0.074$, $n = 10$)

To evaluate the performance of LMS, we determined the μ that resulted in the *optimal* performance for each game play (see Figure 3.13). Compared to the PID-based approach, the percentage of frames missing their deadlines is slightly increased if the LMS predictor was used (at maximum by 1.8% for game play Massive-1). In order to evaluate the robustness of LMS, we then used these game-specific μ's for the prediction of the other game plays. Compared to the PID-based predictor, the robustness of LMS is significantly better, as (i) the maximum deviation from the optimal performance was observed to be 3.99%, whereas for the PID-based approach a deviation of 7.3% was observed, and more importantly, (ii) for no combination of game plays the predictor became unstable. Based on these results we conclude that the LMS predictor is an improved choice for modeling Quake II workloads.

Hardware-based rendering: For Quake II workloads it was possible to observe convergence of the LMS weights. However, evaluating Equation (3.4) for hardware-rendering based workloads revealed that for nearly all game plays a significant percentage of weights did not satisfy this condition. Figure 3.14 shows how the weights evolved over time for the game play Crysis-1. Clearly, convergence of the weights cannot be observed in this case, though the search space of the learning rate and the predictor's order has been exhaustively evaluated.

As convergence is required for an accurate and stable workload prediction, the LMS Linear predictor is not the correct choice for highly varying workloads as in the case of hardware-based rendering. Using the predictor would lead to instability and ultimately our DVFS algorithm would only select the highest and lowest processing frequencies (see Section 3.4.1). To overcome this problem we now introduce a generalization of the LMS linear predictor, the autoregressive moving average model.

3.6 Autoregressive model-based prediction

In literature, an autoregressive moving average (ARMA) model is a generalization of the LMS linear predictor given by Equation (3.1). Note that another generalization of LMS Linear Predictor could have been to use a non-linear model or a linear model with non-linear functions of past workloads as predictors. However, fitting such models is complicated and time consuming and in the following we will show that already the much simpler ARMA model and even a sub-class named AR models suffices to accurately and efficiently predict game workloads.

The ARMA(n, m) model is given by the following equation:

$$\begin{aligned} c[i+1] &= \sum_{k=0}^{n-1} w_k \; c[i-k] \\ &+ \sum_{j=0}^{m-1} v_j \epsilon[i-j] \\ &+ \epsilon[i+1], \end{aligned}$$

where $\epsilon[i]$ are the white noise error terms and $\epsilon[i+1]$ is the error in the linear ARMA representation of the current frame's workload $c[i+1]$. We can compute a prediction based on the workloads $c[i]$, the previous prediction errors $\epsilon[i]$ and the model parameters: The autoregressive coefficients w_k, the autoregressive order n, the moving average coefficients v_j and the moving average order m.

An autoregressive (AR) process is a special case of an ARMA process where the model does not account for past pulses i.e., all v_j's are equal to zero. AR processes are sometimes preferable as they are easier to interpret. Also, using AR models is not as restrictive (compared to using ARMA models) as it might seem, as a large class of ARMA models can be expressed as infinite order AR models, which are known as invertible ARMA models [15].

The ARMA model is appropriate when the system under consideration can be thought of as a *stationary time series process* whose output depends *linearly* on past values, as well as on independent inputs introduced to the system.

3.6.1 Stationarity tests

There is no test that allows directly testing data for stationarity. A common approach is to test for the existing types of non-stationarity using the tests described in the following.

The **Dickey-Fuller test** [33] is well known for testing time series data represented by an autoregressive model. The test checks for the existence of unit roots, a type of linear non-stationarity. Towards this the test assumes the process to be AR(1), and checks whether there is an unit root, which for an AR(1) process equivalent of being non-stationary. This is the simplest test of non-stationarity, but is very limited in nature because of the AR(1) assumption.

The **Augmented Dickey-Fuller test** [33] is a generalization of Dickey-Fuller as it accommodates ARMA(p,q) processes of unknown orders. The differences data $\Delta c[i]$ of the time series is decomposed by fitting the parameters (using ordinary least squares) of the following equation:

$$\begin{aligned}\Delta c[i] &= \mu + k \cdot i + \beta c[i-1] \\ &+ \alpha_1 \Delta c[i-1] + \alpha_2 \Delta c[i-2] + \dots \\ &+ \alpha_p \Delta c[i-p] + \epsilon[i]\end{aligned}$$

where $\Delta c[i] = c[i] - c[i-1]$. The parameters are estimated by performing a linear regression. Under the null hypothesis H_0 that there is an unit root, i.e., the model is non-stationary, all parameters except for μ, i.e., k, β, α_1, α_2, ..., α_p, will be zero. In case the process is not linear non-stationary one or more of the other parameters are non-zero. Said and Dickey proved in [127] that the same test works not only when c[i]s have AR models, but also for ARMA models. We have tested our data as suggested by Said and Dickey for a large value of P to ensure that the data are actually stationary.

In the **KPSS test** [82] it is assumed that time series data $c[1], c[2], \dots, c[T]$ can be decomposed as the sum of a deterministic linear trend, a random walk, and a stationary error:

$$c[i] = k \cdot i + r[i] + \varepsilon[i],$$

where k is a constant, $r[i]$ is a possible random walk (non-stationary) component

$$r[i] = r[i-1] + u[i],$$

where $u[i]$s are white noise with constant variance σ^2 and $i = 1, 2, \dots, T$. In case $\sigma^2 = 0$ and $k = 0$ the process is stationary. The KPSS test therefore has the null hypothesis $\sigma^2 = 0$ and $k = 0$ which it tests against the alternative $\sigma^2 > 0$ or $k \neq 0$. The test considers residuals $e[i]$ from a simple linear regression of $c[i]$ against i, and rejects the null hypothesis for large values of $\sum_{t=1}^{T} \left(\sum_{i=1}^{t} e[i]\right)^2 / \hat{\sigma}_o^2$, where $\hat{\sigma}_o^2$ is a suitable estimate of variance of the stationary part ε_t.

This test happens to be robust against other non-stationary alternatives as well. The alternative hypothesis of this test is a generalization of the null hypothesis Dickey-Fuller and the augmented Dickey-Fuller tests.

All of the tests were applied on the twelve game workloads (both, software- and hardware-rendering based). No test showed any evidence of non-stationary trend in the game workload data. Hence, we conclude that the data are stationary and ARMA and AR models can be applied.

In the case of the LMS linear predictor, the gains w_k change over time as they are learned online based on the learning rate μ and Equation (3.2). Under high workload variations, such online systems might adapt to the data too quickly and therefore can be misled (resulting in unstable behavior). In contrast, for ARMA and AR, the set of parameters is fitted offline only once as described in the following, and does not change over time.

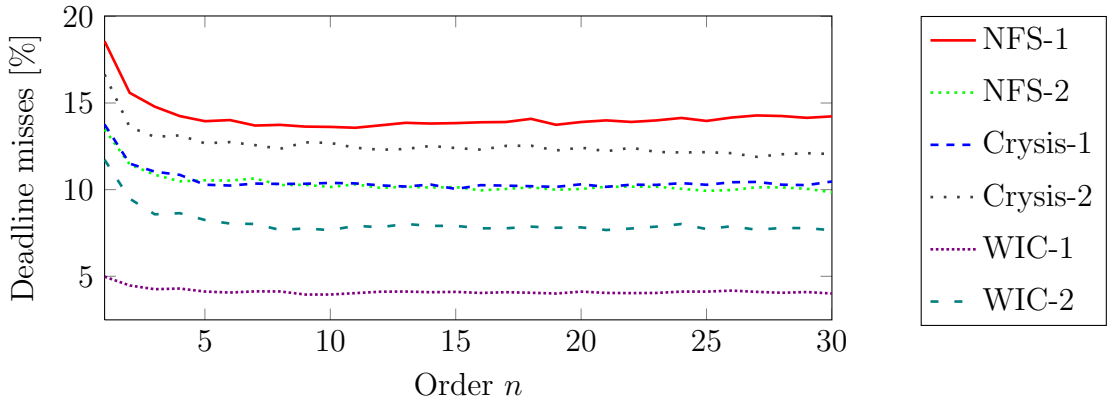


Figure 3.15: Percentage of missed deadlines for different orders n of AR(n)

3.6.2 Fitting ARMA and AR models

Before the ARMA and AR models can be used to predict the game's workload, suitable model parameters ($w_0, \ldots, w_{n-1}, v_0, \ldots, v_{m-1}$) and model orders n and m have to be determined that guarantee a good prediction performance.

Model parameters: The ARMA model parameters can be fitted using a maximum likelihood method that maximizes the joint probability function of the model given the data. We chose to perform this maximization through a subspace method using an iterative Gauss-Newton algorithm (using the system identification toolbox from MATLAB) [92]. For large data sets this approach is usually slower in implementation compared to the steepest gradient method employed by the LMS filter. However, determining these parameters needs to be done only once and is done offline based on pre-recorded game workloads. As we discuss later, once these parameters are determined, they remain constant not only across game plays, but also across different games.

A common approach to tune parameters of AR models is the non-iterative, least-squares method [99] which is as well used by the MATLAB System Identification Toolbox. All results presented in this work are based on models which have been tuned using this toolbox.

Model order: Using the methods described above, we trained the model parameters for each DirectX-based game/game play individually with $n,\ m\ \leq 100$. Figure 3.15 shows the gained results for an AR(n) model. We observed that an order greater than 10 for both, n and m only slightly improved the predictor's performance. Hence, for the sake of simplicity and to avoid an increased algorithm's complexity, we restricted the remainder to orders $n, m \leq 10$. The best performance for $n,\ m\ \leq 10$ was achieved with an ARMA(10, 9) model. It could be observed that the performance of an AR(10) model provides a very good fit which is close to the ARMA(10, 9) model, and their forecasts are also close. Hence, for the sake of simplicity and in view of the previous discussion, we restrict the remainder of our analysis in this work to AR processes of order 10.

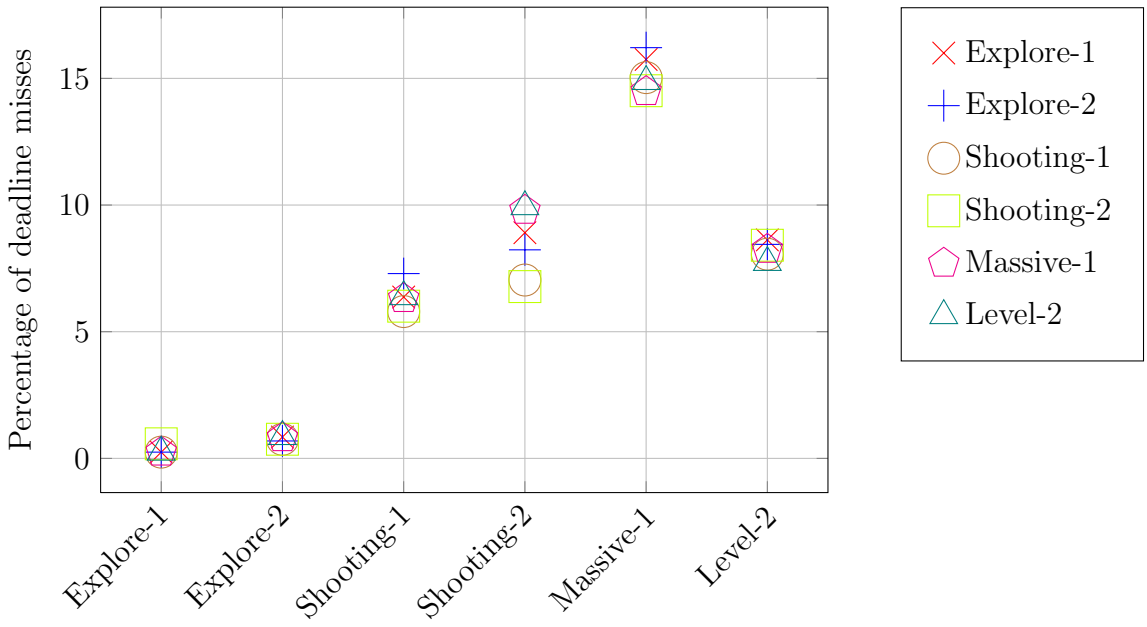


Figure 3.16: Performance evaluation of game-play-optimized AR(10) model parameter sets (with software rendering setup)

3.6.3 Evaluating AR models

Figures 3.16 and 3.17 depict the performance of the AR(10) models for the software and hardware-rendering approaches respectively. For each game and game play, a model was individually derived with the MATLAB toolbox. The model was then tested with the corresponding game play and with other games and game plays. It can be seen that the best performance achieved with AR(10) is approximately the same as when using an individually tuned PID controller. In case of an AR model, however, no instability has been observed as it can happen for the PID controller. As can be seen in Figure 3.17, only a small performance difference (at maximum 0.68 %) can be noticed when the model, tuned using the workload from one game or game play, is used for the workload prediction of a different game or game play. In comparison, with the PID-based prediction a significant difference of up to 12.6 % of more frames missing their deadline is observed (see Figure 3.9). The performance difference is close to zero if the models are tuned on a per game basis, i.e., using the individually tuned NFS-2(+) for NFS-2 itself results in 10.3 % of deadline misses. Using NFS-1(×)-based models for NFS-2 results in 10.5 %.

We conclude that game workloads are accurately and efficiently represented by the AR models we have fitted and no further generalization (e.g., to non-linear models) as described at the beginning of Section 3.6 is required. We conclude that AR models are robust enough to workload variations such that the model can be fitted only based on one sample game play. As shown, the resulting model will provide a good performance, even if used for game plays which it has not been optimized for. A further generalization (e.g., to non-linear models) as described at the beginning of Section 3.6 is not required.

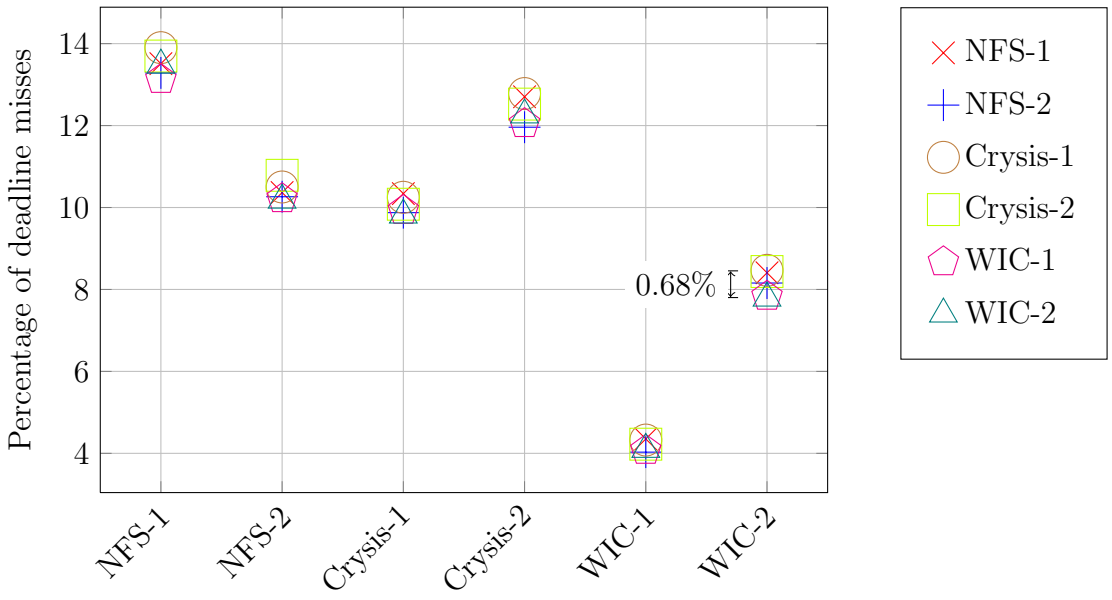


Figure 3.17: Performance evaluation of game-play-optimized AR(10) model parameter sets (with hardware rendering setup)

3.7 Power measurement results

So far, the evaluation of the predictors' performances focused on the prediction quality in terms of percentage of frames missing their deadline. The evaluation suggests the suitability of autoregressive models for game workload prediction. In addition to game quality i.e., minimizing the number of frame deadline misses, we are also concerned with minimizing the average power consumption of the processor. In this section we present the experimental setup and results of the overhead and power measurements based on the proposed DVFS algorithms. We show that power savings of up to 35.8 % can be achieved while maintaining a desired gaming quality.

3.7.1 Power management overhead

Any power management technique involves overhead, which in our case consists of the time consumed for the application profiling, the computation time for the workload prediction and the settling time of the voltage and frequency regulator. In case of the software-rendering based setup, the source code has been directly instrumented and the profiling overhead merely consists of reading the time stamp counter (TSC) values. This overhead can be completely neglected when compared with a game frame's workload. For the hardware-rendering based setup, all DirectX calls are intercepted. On an average we observed an overhead of 2.34 ms per frame. With a target frame rate of 40 fps this equals to 9.3 % of the available time per frame. As we will show in the following, despite this overhead we can achieve considerable power savings. The overhead is mostly caused by

the interception itself as every Direct3D command first calls our proxy library before being forwarded to the original Direct3D library. It could be reduced to negligible levels if the DVFS algorithm was incorporated directly into the DirectX graphics API or if the API offered an interface. As a consequence the power savings could be even higher.

Table 3.5: Overhead of the four different workload predictors

Predictor	# of Cycles
PID	15.0
LMS(10)	323.9
AR(10)	26.3
ARMA(10,10)	52.8

The overhead for the different workload prediction techniques is given in Table 3.5. Compared to the number of cycles that are at least available to the CPU per frame this overhead can be completely neglected.

The overhead for scaling an Intel® Core™2 Duo E6850 processor is available from literature with 30.3 μs at maximum [114]. If we assume frequency switches at 40 Hz this scaling overhead makes up 0.12 % of a frames processing time.

3.7.2 Power savings with software rendering

In the software-based rendering setup (used in Quake II), the power measurements were performed at the output of the laptop's AC Adaptor. A Texas Instruments MSP430 microcontroller was employed to measure both, the DC voltage $v(t)$ and the current $i(t)$ with the help of a shunt resistor and an amplifier. The average power consumption $\overline{P}$ was then calculated according to

$$\overline{P} = \frac{1}{l} \sum_{t=0}^{t=l} v(t)i(t)T,$$

where l is the duration of the game and $1/T$ corresponds to the sampling rate, which was set to 1kHz. The microcontroller was operated via a serial connection from the laptop and the power measurements were logged for every game play. The control interface for operating the microcontroller was also integrated into the Quake II source code in a way that the measurements through the controller could be started and stopped at the beginning and the end of a game play. In this manner, we ensured synchronization between the start and the stop of the game play and its corresponding power measurements.

This setup provides measurements corresponding to the power consumption of the entire laptop. During the measurements the battery was removed from the laptop to avoid measuring the power consumed for re-charging the battery as well. Additionally, we ensured that the laptop settings remained constant during all measurements (i.e., we maintained

Table 3.6: Average power consumption for available frequencies

Frequency [MHz]	800	1066	1333	1600	1866
Power [Watt]	21.1	23.3	25.8	29.1	33.0

Table 3.7: Power consumption of PID-based workload predictor and LMS linear predictor for different game plays. The savings are given in percent compared to running the CPU at highest frequency

Game Play	Average Power Consumption [W]			Savings using AR [%]
	$\overline{P}_{PID}$	$\overline{P}_{LMS}$	$\overline{P}_{AR}$	
Explore-1	21.42	21.6	21.2	35.8
Explore-2	21.71	21.9	21.6	34.6
Shooting-1	23.1	23.6	23.0	30.4
Shooting-2	25.2	24.8	25.1	24.0
Massive-1	25.1	23.9	25.9	21.8
Level-2	23.7	23.2	23.7	28.3

the same settings for display brightness, switched off the wireless LAN and removed all the devices connected to the laptop except the microcontroller used for measuring power consumption).

Table 3.6 shows the laptop's average power consumption for all available frequencies of the processor. In our simulation, these recordings were used to approximate actual power consumption. As we measured the system's total power consumption, our measurements included the power consumption of, for example, memory or front-side bus, which highly depends on the load of the system [137]. Therefore, we acquired power measurements for each utilized Quake II demo and all available frequencies together with the corresponding workload profiles. A maximum variation of 2.4 % in average power consumption was observed. We conclude from this data that 36 % of the total power (compared to maximum power consumption of the laptop) may be saved at the maximum by running the system at the lowest frequency at all times. This, however, will result in an unreasonably high percentage of deadline misses, significantly reducing the game's quality.

We incorporated all the proposed predictors into the Quake II source code. Table 3.7 gives the measured average power consumption for each game play using the individually tuned PID-based predictor ($\overline{P}_{PID}$), the LMS Linear Predictor ($\overline{P}_{LMS}$) with a μ of 0.074 and individually tuned AR(10) models ($\overline{P}_{AR}$). The observed difference among the different predictors in terms of average power consumptions is negligible. The last column shows the achieved power savings of AR(10) compared to the power consumed if the laptop is clocked at the highest frequency. It may be noted that between 24.0 % and 35.8 % of power is saved depending on the characteristics of the game play.

Note that as presented in Section 3.4.2 and depicted in Figure 3.7, a non-optimal choice of PID gains affects the percentage of frames missing their deadline significantly more than the average power consumption. The cross-validation for PID and AR(10)-based predictors experimentally confirmed this behavior. Only small variations in respect to the optimal average power consumption were observed: 2.53 % and 1.34 % for PID and AR(10) respectively. For each game play, not only the power savings were similar, but also the game quality which can be seen from Figures 3.8 and 3.16. However, as shown in Section 3.6 it is sufficient to tune the AR model only once, whereas the PID gains need to be tuned for each game play individually to avoid large performance drops. This individual tuning is not practical in a real-life scenario.

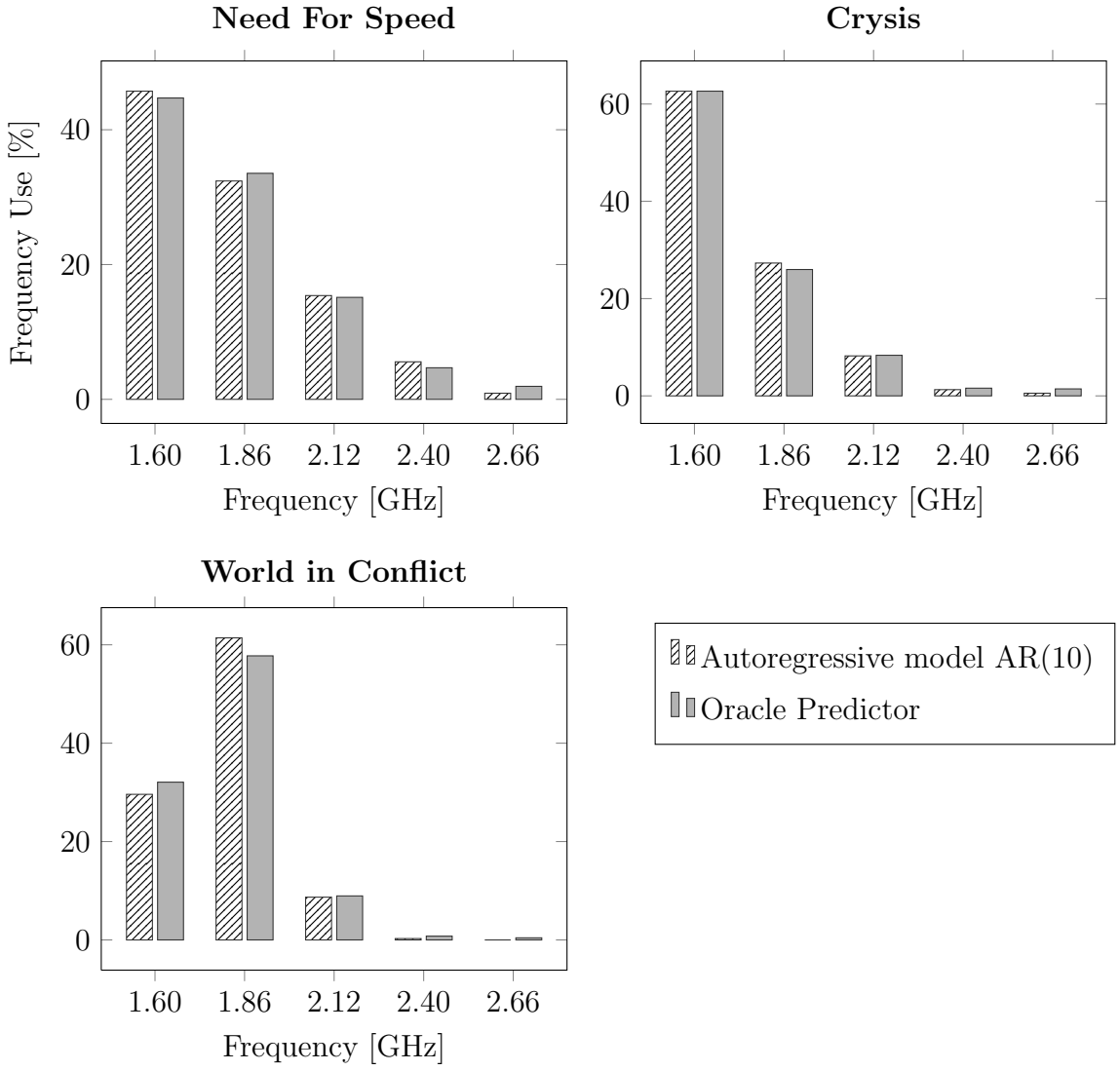


Figure 3.18: Frequency histogram of Need for Speed, Crysis and World in Conflict using autoregressive models AR(10) and an oracle predictor (perfect prediction)

3.7.3 Default Linux power management

Linux is equipped with a widely-used Ondemand Governor [110] for power management. We ran the Quake II game plays with the Ondemand Governor (with default settings) enabled and logged the current frequencies, workload profiles and average power consumption. We observed that with the Ondemand Governor, it is possible to obtain approximately 7 % power savings (for all game plays), whereas the AR-based Predictor achieves power savings of up to 35.8 %. As Quake II is programmed as an endless loop, the Ondemand Governor will always detect high system utilization. Consequently, voltage/frequency scaling cannot be enabled during most of the time.

3.7.4 Power savings in case of DirectX-based games

For the DirectX-based approach we incorporated the proposed predictors into the proxy DLL (see Section 3.3.2). As both, the PID-based Predictor and the LMS linear predictor were not stable the measurement results are not shown here. Figure 3.18 gives the ratio at which the available frequencies of the processor were used for the three games Need for Speed, Crysis and World in Conflict. The results are given for the autoregressive model and an *oracle* predictor. The oracle predictor is a purely theoretical construct and is assumed to know the full future (see Chapter 5). Hence, it gives the maximum possible power savings with the smallest possible number of frames missing their deadlines. The AR(10) model in comparison selects the frequencies with almost the same distribution. This establishes that these policies result in a close to optimal energy consumption of the processor.

3.8 Summary

With the aim of DVFS-based power management, in this work we have proposed workload prediction schemes for game applications whose parameters need to be tuned once during an *offline phase* and may then be used for both – game plays, as well as new games that are not a priori known. Hence, our focus has more been on the *robustness* of the prediction scheme, rather than its optimality. In other words, occasionally, a predictor that is hand-tuned for a particular game play might outperform our proposed offline-tuned predictor. However, the overall prediction quality of the offline-tuned predictor was comparable to a hand-tuned predictor.

Towards this, we studied several games – Quake II, whose source code is available, as well as more modern closed-source games like Crysis, Need for Speed (NFS) and World in Conflict (WIC). We showed that concerns about stability and tedious hand-tuning of parameters make known PID-based workload prediction schemes unsuitable for any of the considered games. We next studied a LMS linear predictor, which worked well for

Quake II with software rendering, but not for Crysis, NFS or WIC that rely on hardware-rendering support. For these games, we showed that the autoregressive moving average (ARMA) model and its simplified version, the AR model, address the concerns that arise in a real-life power management setup, viz., that not all game plays and game applications are a priori known. Our results are consistent across all evaluated games and particularly attractive because parameters of an AR-model, tuned using a set of game plays, provide good gaming quality as well as power savings when applied not only to different game plays, but also different game applications. The similarity between results obtained using AR and ARMA also show that more complex non-linear models or linear models with non-linear functions of past workloads are not required.

Further, we have exploited the software (in particular the graphics processing) architectures of modern games by intercepting API calls made by the game application to the graphics library. These were used to estimate the execution times of game frames, which were then used for workload prediction. This enabled us to apply our power management scheme to a variety of modern closed-source games, whereas all previously known techniques required modifications of the game's source code which is not available for most modern games. The applicability of our scheme to closed-source games ensures its practical relevance. Further, our techniques differ from workload prediction and DVFS schemes known for video processing in two major ways. First, we showed that PID controllers that were successfully used for video applications, are not practical for game applications, which require different workload prediction schemes, such as the ones proposed in this chapter. Second, we exploited the graphics processing architectures of games (through API call interception), that do not arise in video processing. It is possible to combine our results with frame-workload prediction schemes that analyze the contents/structure of game frames as proposed in [51], where a PID-based scheme was combined with frame structure analysis. However, workload prediction based on analyzing the contents of a frame is mostly relevant for software-based rendering schemes. This is because only the rendering (and not AI or game physics) workload may be estimated based on the contents of a frame. Hence, such schemes will not be relevant for more modern games that rely on hardware support for rendering.

For popular mobile Android games we have observed that a significant amount of time of the total gaming time is not spent in the actual gaming phase, but for example in level selection or highscore menus. In the next chapter, we first show how game power management can be implemented on Android-based platforms to then describe a mechanism for the detection of different gane states. Based on this game state detection we develop game state-specific power management strategies that allow a significant reduction in terms of total power consumption.

4

State-specific power management for closed-source games

In the previous chapter, we have described a power manager that predicts the future workload of games and scales the processors voltage and frequency with the goal to reduce the power consumption while maintaining a constant target frame rate. While in the previous chapter we used DLL injection to apply the developed power management techniques to closed-source Windows games, this chapter details, how this approach can be extended to Android-based mobile platforms and popular closed-source mobile games like Cut the Rope and Temple Run.

Games have different states like the loading, main menu, level selection menu, movie and gaming state. Particularly for mobile games, we observed that a substantial amount of time is *not* spent in the actual gaming state, e.g., 50.8 % of the time for a typical five minute game play of Cut the Rope were spent in other states. This results in a substantial amount of energy that is consumed not only during the highly interactive gaming state, but as well during the loading, movie and menu state. Each of these states has different requirements and shows different workload characteristics, e.g., the loading state is likely to be memory bound and the menu state, due to lower interaction, is likely to require only a lower frame rate compared to the gaming state to satisfy the user. In this chapter we introduce a technique that allows detecting the current state of a game. Using this knowledge, we employ state-specific power management that exploits game state-specific workload characteristics.

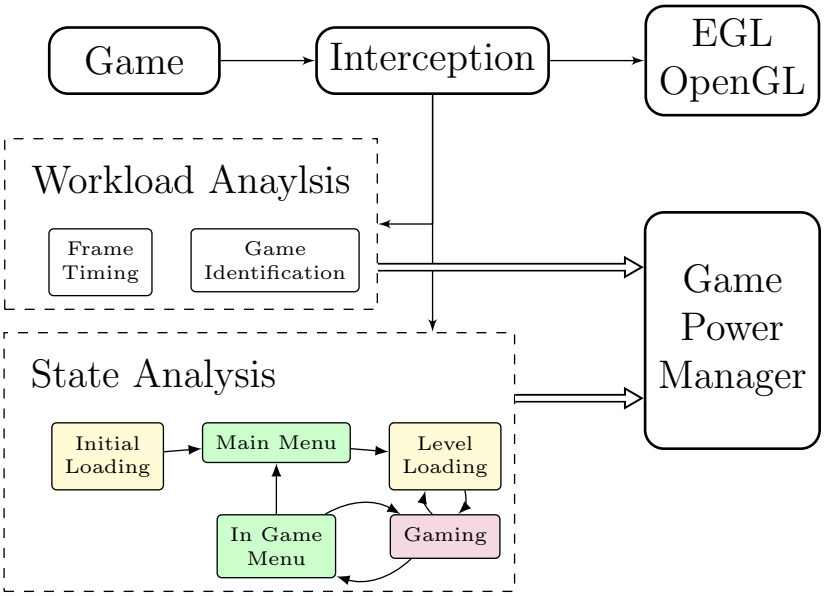


Figure 4.1: System architecture overview

The **contributions** of the work presented in this chapter are as follows:

- The experimental setup of this chapter is based on the Samsung Galaxy Nexus smartphone as for this phone it was possible to change the Android OS to our needs. The phone has been physically modified to allow detailed profiling and accurate CPU power measurements. This, for the first time, allowed to accurately analyze the CPU power consumption for highly interactive game applications on a frame-by-frame basis. The results are presented for three of the most popular Android games, namely Jetpack Joyride, Cut the Rope and Temple Run.

- While in the previous chapter, DLL injection was used to intercept the communication of the game with the GPU on Windows-based platforms, here we describe the interception of graphics calls in more detail and explain how this approach can be implemented on Android-based platforms by intercepting graphics calls to Android's graphics libraries, namely EGL [78] and OpenGL ES [79]. Based on this setup, we developed the `generic game` governor, following the principle described in Chapter 3. It receives the frame timing information (see Figure 4.1) and predicts the game's future workload using an autoregressive model-based predictor (see Chapter 3). This governor is not state-specific and maintains the same constant frame rate for all states and for any game. We show that substantial power savings of up to 32.4 % are possible compared to the default Android `interactive` governor.

- We propose a technique that allows to accurately detect the current state of a game such as the loading, menu, gaming and movie state. The developed technique is widely applicable as it does not require to modify the game's source code. Using this detection, we for the first time present a detailed analysis of CPU power consumption during different states of popular Android games. We show that, contrary

to expectations, a significant amount of time and energy is spent in states other than the actual gaming state. This motivates a game state-specific power management strategy.

- To reduce the power consumption of individual states, we propose a state-specific power management that, amongst others, exploits memory bound phases and maintains state-specific target frame rates. We extended the `generic game` governor with the game state detection technique and for example reduce the target frame rate during the menu state or select a constant processing frequency during memory-bound loading phases. Using this `game state specific` governor, we achieved power savings of up to 26.9 % compared to the `generic game` governor and 43.2 % compared to the `interactive` governor.

- Both, the `generic game` and the `game state specific` governor significantly reduce the smartphone's power consumption, but at the same time increase the percentage of frames missing the deadline. To clarify if the observed increases can be actually noticed by the user, we have performed a user study. The results of the study indicate that there is no perceivable difference between Android's default governor and the power managers suggested by us.

Related work: Only few work leverages detection of game internal states for power management. As described in Chapter 2, the work presented in [6,7,149] uses information about the player's position and velocity to switch the power state of a wireless network interface card (WNIC). All of the work is based on Quake III, due to its source code being freely available. While in [6, 149] the game's source code has directly been modified, [7] proposes an API that could be used in future by game developers to forward relevant information to the WNIC power manager. The work of Pathania et al. [115] detects different game states of the game Asphalt 7 based on CPU and GPU utilization transitions: During the loading state of the game, it has been observed that the GPU utilization is significantly lower compared to the gaming state, while for the CPU the opposite was true. For the games used in this work, we could as well observe significant CPU utilization changes between loading and gaming state. However, such changes were not observed during the menu and gaming state. In this chapter, we will present a texture-based algorithm that allows to differ on a fine-grained basis between gaming, menu, loading and movie state and hence allows applying a finer-grained state-specific power management. Pathania et al. further suggest to raise the CPU frequency to the maximum during loading states to shorten the loading time to the minimum possible. In Section 4.5 we will show that, for the games used in this study, such increase of the processor's frequency should be avoided since it shortens the loading time only to a small extent while exponentially increasing the power consumption.

Organization of the chapter: Section 4.1 presents the Android-based system architecture, explains details about Android's graphics architecture and our Android-specific modifications. Section 4.2 briefly recaps the `generic game` power manager and the autoregressive model-based prediction. Section 4.3 describes how a detailed profiling of the game's communication with the OS allows identifying the game states. This is followed by

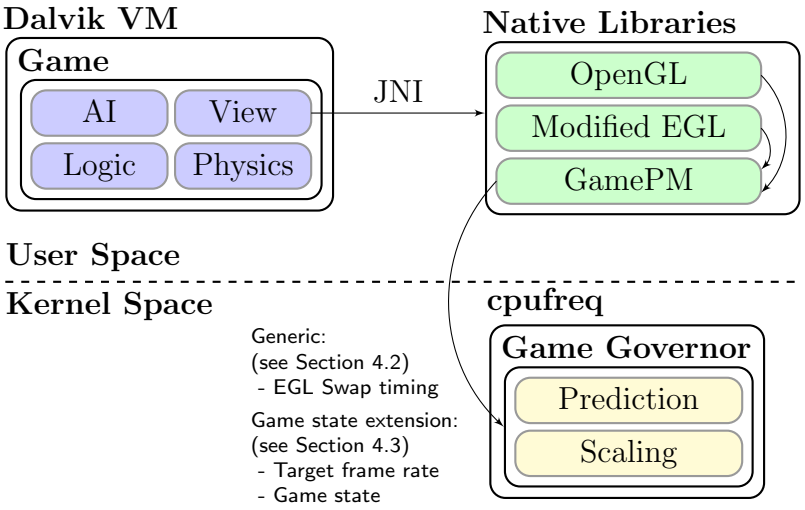


Figure 4.2: Proposed interception mechanism

a description of the experimental setup in Section 4.4. Here, we present details about the required modifications to the Samsung Galaxy Nexus, the experimental setup on which our measurement results are based on, the selection of games used for this study and the experimental methodology. In Section 4.5, we present the power measurement results obtained using both, the `generic` and the `game state specific` governor. We confirm, based on a user study, that there is no noticeable difference between the power manager proposed by us and Android's default governor. In Section 4.6, we propose an API that might be used by the game developer in future for state-specific game power management. Section 4.7 concludes this chapter and gives an outlook for future work.

4.1 Proposed system architecture

Figure 4.2 gives an overview of the proposed system architecture towards game state-specific power management. We illustrate the architecture using OpenGL ES and the Android operating system. The technique can as well be applied to systems that use the DirectX (see Chapter 3) or OpenGL API to render game content, as the basic mechanisms, we leverage for our power management, exist in these APIs as well.

4.1.1 Android graphics 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 making calls to native C/C++ libraries. The game uses this interface to call native OpenGL ES and EGL functions. Calls are then forwarded to the GPU driver and finally

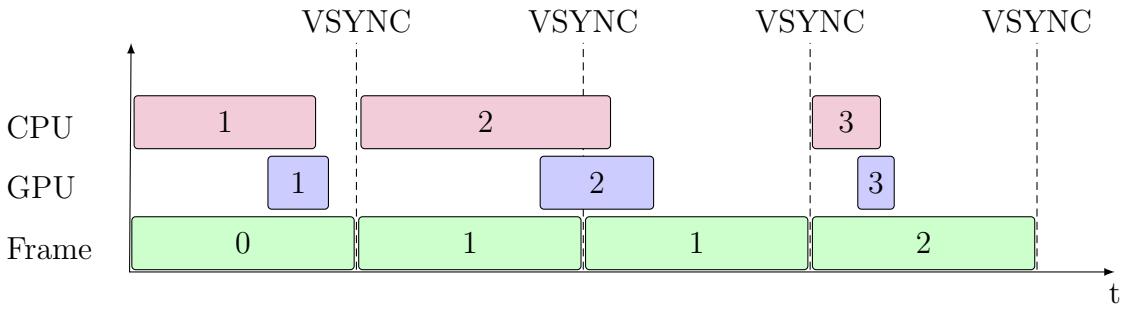


Figure 4.3: Typical frame timing

to the GPU where the content is rendered to the so-called *back buffer*. Similar to DirectX, the game calls a particular function, (i.e., `eglSwapBuffers()`in the case of OpenGL ES), once the game finished all computations for the current frame and issued all the required render calls. This will cause the GPU to swap between the front and back buffer.

The Windows-based presented in Chapter 3 was configured such that a continuous processing of frames was performed by the game. In Android the processing of frames is synchronized to the display refresh. A typical timing diagram of this is shown in Figure 4.3. The processing of Frame 1 starts with the vertical synchronization signal (VSYNC) signal from the display. As described, at some point OpenGL calls are issued by the game view and the GPU starts rendering. After `eglSwapBuffers()` has been issued, the CPU waits for the next VSYNC signal. Once the monitor reads the buffer content and signaled this to the CPU, the processing of Frame 2 starts. The total processing time of a frame is directly influenced by the processing speed of the CPU and GPU. In some cases, the processing time might be longer than the display's refresh interval like shown in Figure 4.3 for Frame 2. As a result, the previous frame is shown twice (in this case Frame 1).

Due to this display synchronization, an upper bound of the frame rate is given by the display's refresh rate which is 58 Hz for most mobile devices. All results shown in this chapter are based on a target frame rate of 58 frames per second except otherwise specified. We show that even for this frame rate considerable power savings are possible. Note, that for many games a lower frame rate might already be sufficient and hence even more power could be saved. In future, each application could specify its desired target frame rate itself using the later proposed interface.

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.

4.1.2 Power management-specific modifications

To predict the future workload and apply power management schemes as described in Chapter 3, the frame timing needs to be measured. In Android, we leveraged the above described OpenGL ES interface and instrumented the `eglSwapBuffers()` function directly in the source code of the EGL library. Each time this function is called, we recorded an accurate time stamp which then was used as input to our workload prediction. Scaling the processor's frequency is only allowed in kernel mode. Towards this, we have modified the Linux `cpufreq` module and extended it with our own Android power management governor. At the load time, the modified `cpufreq` 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. All of the following calls use the opened node to send the recorded time stamps to the governor via `ioctl` syscalls. The governor receives the time stamps and performs the workload prediction like described in Section 4.2 and shown in Figure 4.2. 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. The desired target frame rate can be configured using the device node to the `cpufreq` module.

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 entry in `/proc/pid/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 `interactive` governor.

For the second part of this work we have instrumented and redirected specific calls of the OpenGL library. The information, gathered from these calls, was used to determine the current game state (see Section 4.3). State change events and state-specific details like the target frame rate were forwarded using the above described device node interface to the `cpufreq` module.

In addition to the interfaces required for the online phase of the power management, we have implemented several stages to log required game timing data and OpenGL call parameters. This allowed us, amongst others, to analyze timings and tune the prediction mechanisms offline using MATLAB as described in the following chapter.

4.2 Generic game power management

As described in Chapter 3, the workload of a game's next frame is predicted based on the workload of previous frames. We have carried out experiments using several closed-source Android games and confirmed the findings of Chapter 3: An AR model of order $n = 10$

Table 4.1: AR weights scaled by a factor of 1000

Game	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9
Cut the Rope	109	142	90	80	82	81	92	67	136	102
Jetpack Joyride	160	112	125	129	107	93	63	79	70	24
Temple Run	159	185	121	106	99	89	72	81	49	18

which is tuned offline and once per game was found to provide an accurate workload prediction. Table 4.1 gives the resulting AR weights for the three games used in this chapter. Despite significant differences between optimal weights per game, the AR model-based predictor was observed to be robust accross different games. For our experiments, we still used weights that have been tuned per game to guarentee the best prediction performance. If the game and hence optimal weights are not known, one possible approach could be to tune the weights of the autoregressive model based on timing information that is recorded during an initialization run. During this run, Android's default `interactive` governor or default weights could be used. Once the optimal weights are derived based on the recordings, Android could switch to the `generic game` governor.

4.3 Game state-specific power management

The `generic game` governor presented in the previous section targets to maintain a constant frame rate during the whole duration of the game. All games consist of typical states, like the gaming state, the state during which the user navigates through the game menu or the loading state of the game. Each of those states might benefit from state-specific power management strategies. For example, the memory bound phase during the loading state can be exploited by reducing the CPU's frequency without impacting the performance. Due to differences in terms of the user interaction rate, in other states a reduced frame rate might already be sufficient to satisfy the user. In the following, we propose four different power management schemes, describe the observed game states and how they are mapped to the corresponding schemes. Note that the proposed schemes are only one possibility of power management for the corresponding states and different schemes could as well be applied.

4.3.1 Definition of game states and power management schemes

Figure 4.4 shows an example of a possible game state transition diagram. Not all games considered in this work have all of the states or have the same transition possibilities. For example, the games Jetpack Joyride and Cut the Rope have no movie scene. Further, we define four basic power management schemes: *Loading*, *No Interaction*, *Low Interaction* and *High Interaction*. Each state of a game is mapped to one of these power management schemes as described in the following.

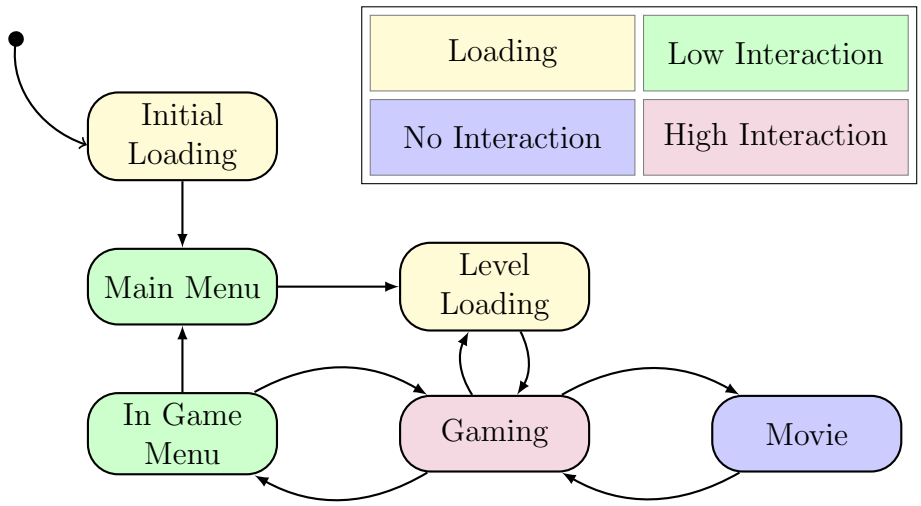


Figure 4.4: Example of possible state transitions and the mapping of the states to their corresponding power management strategies

Loading: During a game typically two loading states occur, the initial loading of the game and the loading of the game level. During the initial loading state the application pre-loads textures, audio and other objects relevant for the main menu scene. Typically, there is only a very light-weight graphics representation in order to finish this state as quickly as possible. After the player has selected a level, the level loading state might be entered during which level-specific data is loaded. Here, often a more complex graphics representation is used to shorten the user's impression of the loading times. Both, the initial and the level loading state are likely to be memory bound which means the CPU is waiting to a large extend for data. As will be shown in the experimental section, it is therefore possible to lower the processor's frequency without prolonging the loading time.

No Interaction: Many games have states during which movies are replayed, e.g., Temple Run shows an animation after the initial loading state. This movie state does not allow any user interaction (except from skipping the movie) and is therefore mapped to the *No Interaction* scheme. We reduced the target frame rate from 58 to 20 frames per second which still gave the impression of a smooth video playback in the case of Temple Run.

Low Interaction: Once the required objects have been loaded, the game enters the main menu state. This state allows the user to walk through the menu to, e.g., change options, show the game help or the credits. Further, this state typically includes a view that allows the level selection. We expect this state's processing requirements to be rather low as mostly only graphics animations of intermediate complexity are used and no complex AI or physics computations are required. The main menu state is mapped to the *Low Interaction* scheme as no fast interaction is required like during the gaming state. Similar to the main menu, the in-game menu has low requirements in terms of responsiveness and can be mapped to the *Low Interaction* scheme. However, the in-game menu might be rendered on top of the current scene (like in the case of Cut the Rope). Therefore, the scene

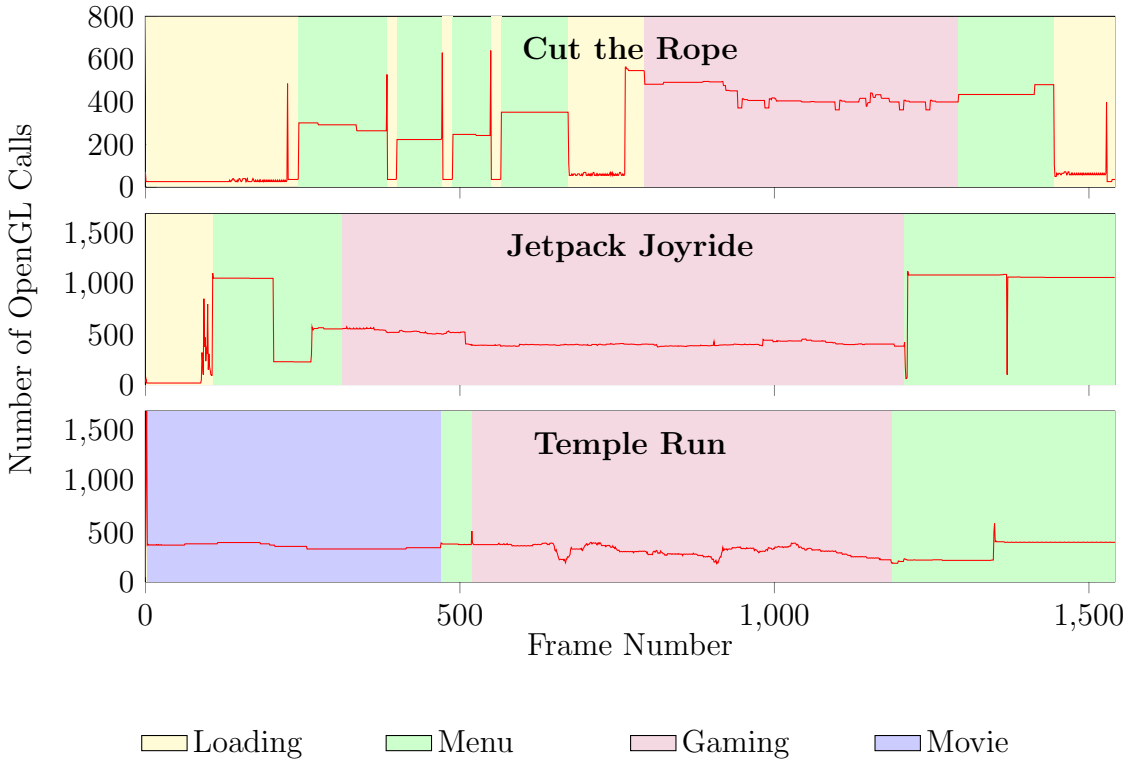


Figure 4.5: Number of OpenGL calls made per frame and the corresponding game states

complexity might be similar to the gaming state. As the degree of interaction is lower during a menu, it is possible to reduce the target frame rate without impacting the gaming quality. We have chosen a target frame rate of 30 frames per second, as in this case, for none of the used games a difference compared to 58 frames per second could be noticed.

High Interaction: Once a level is selected and the game is loaded, the game enters the gaming state. This state typically has the most critical requirements in terms of CPU and GPU processing time and responsiveness. It is therefore mapped to the *High Interaction* scheme. During this state we set the target frame rate to the maximum possible 58 frames per second to guarantee a good gaming experience.

Before the presented power management schemes can be applied, a mechanism is required to detect the game states during the run-time of the game.

4.3.2 Detection of game states

As the source code of the games used in this study is not available, we developed a detection mechanism which is based on the communication of the game with its environment. First experiments were based on the number of OpenGL calls made per frame. Clearly, rendering a frame of a complex game scene is likely to require more calls than a simple

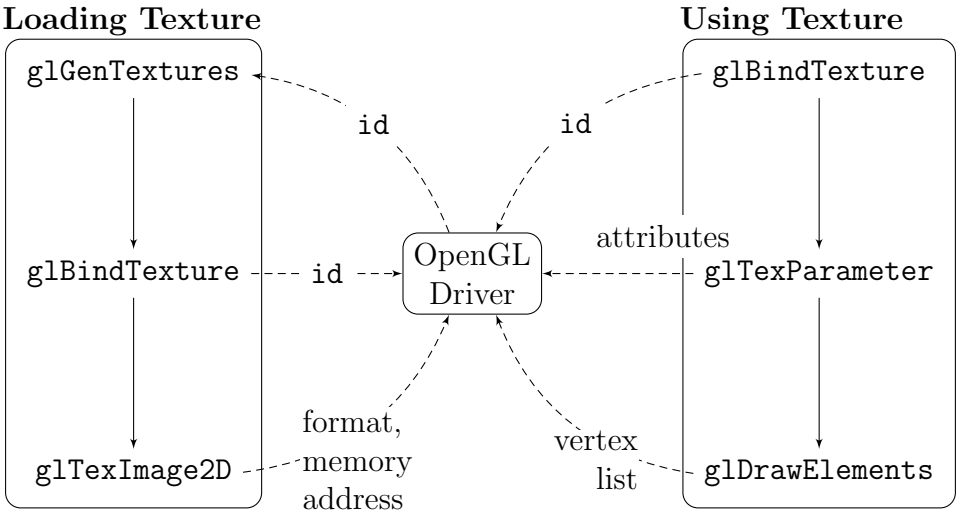


Figure 4.6: Steps required for loading and using a texture including the data flow

loading or main menu screen. First experiments were based on the assumption that complex in-game frames are likely to require a higher number of OpenGL calls than simple loading or main menu frames. In this case, the number of OpenGL calls per frame could be used to differentiate between the game states. We have modified the OpenGL library to count the total number of OpenGL calls made during processing a frame. Figure 4.5 depicts a resulting graph visualizing the number of OpenGL calls on a frame basis and the corresponding game states. For the games Cut the Rope and Jetpack Joyride a correlation between the number of OpenGL calls and the current game state can be noticed. As expected, during the loading phase only a low number of calls can be observed whereas the number of calls made during the gaming phase is significantly higher for Cut the Rope. However, in the case of Cut the Rope the in-game menu state cannot be told apart from the gaming state as there is no significant difference in the number of OpenGL calls made. For the game Temple Run, it is only possible to detect the loading state with its very large number of OpenGL calls made during the first three frames. A more detailed analysis revealed that a large number of texture loading calls are issued while only a very simple texture is shown as visualization of the loading state. For the other states no clear correlation could be noticed. A more detailed analysis of state-specific OpenGL or system call patterns might allow to distinguish between game states. However, for this approach it is required to perform a feature extraction and pattern matching which is extremely time consuming and has to be repeated for each game individually as it highly depends on how the game is programmed. Instead of this approach, we in the following describe a texture-based state detection that is easy to tune, extremely accurate and highly portable.

Each game uses particular textures for its states, like textures of menu buttons in the menu state or a sand watch texture during the loading state. If one can therefore identify which textures have been used during the frame, the game's state can be detected. In order to draw an object in OpenGL using a particular texture, first a texture object has to be created (see Figure 4.6). This is done by calling `glGenTextures` which returns a

unique `id` identifying the texture object. Next, the generated texture object is bound using `glBind- Texture(id)`. This informs the driver that the next calls relate to that particular object. Before the texture can be used for drawing, the object has to be filled with content. For this, e.g., `glTexImage2D` is called, handing over the format in which the texture resides in memory (size, pixel format, etc.) and a pointer to the texture's location in memory. The content from this location is then copied into the object's texture buffer and the texture is ready to be used. Typically, the described initialization of texture objects is done during the loading state of a game. If the game later wants to draw objects using the texture it simply binds the texture object by calling `glBindTexture(id)` before setting the texture drawing attributes and issuing OpenGL drawing calls like `glDrawElements`.

The only parameters handed over to the OpenGL driver related to the texture itself are the texture format and its address in the memory. Both parameters, same as the loading order and the `id` of textures may vary from game run to game run and therefore cannot be used for a distinct identification of the texture. The only constant in this process of texture loading and usage is the image data of the texture itself. To avoid storing large amounts of data for a later comparison we computed an unique hash key according to [128] for each texture getting loaded.

To identify which texture belongs to which state we need to assign states to the computed hash keys. Towards this, we have modified the OpenGL library to store all textures to the mobile phone's flash memory if a particular flag was enabled. This allowed us to view the textures offline and assign states to the corresponding texture hashes. The thereby created `<hash,state>` map is stored to the mobile phone and loaded at load time of the game. If now a texture gets loaded e.g., using `glTexImage2D`, the hash key is recomputed over the data of the texture and looked up in the `<hash,state>` map. If the hash key is found, an entry in a dynamic `<id,state>` map is created. When the game later binds a texture using `glBindTexture(id)` we look up the corresponding state in the `<id,state>` map using the `id`. Note that the `<hash,state>` map only consists of texture entries that clearly identify states. All other textures can be ignored. Hence, the amount of textures that need to be marked during the offline phase reduced to only a small number (in the case of Cut the Rope, only 15 textures were required). The hash key has to be computed for all texture loading calls (like `glTexImage2D`), but not every time a texture is used. Using this method, an extremely robust state detection can be created for a game within minutes.

Note that the above described method for detecting game states requires identifying textures for each game and game state. With the large number of games available in the different stores, this might be not possible for all games. In case the game is unknown, the `generic game` governor described in Section 4.2 could be used. Another solution to this problem might be an API which allows the game developer himself to provide the game state information (see Section 4.6).

In the following, we present the experimental setup and methodology including the measurement setup, the selection of games used for the study, details on texture identification and the workload characteristics of the games.

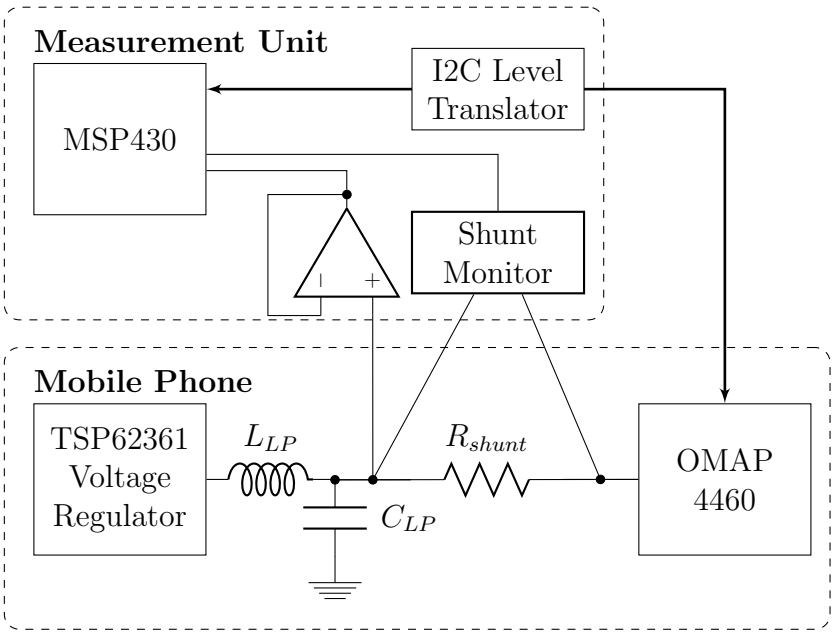


Figure 4.7: Schmatic of the measurement setup

4.4 Experimental setup

For all our experiments presented in this chapter we used the popular Samsung Galaxy Nexus smartphone as for this phone it was possible to customize the Android operating system. The phone hosts a Texas Instruments OMAP4460 [147], a dual core ARM Cortex-A9 1.2 GHz mobile processor. The core of the OMAP4460 processor is the microprocessor unit (MPU) subsystem. It consists of the two Cortex A9 cores, each with a dedicated L1 instruction and data cache, a NEON and a VFPv3 (floating point) unit. The two cores share a 1 MByte L2 cache, general purpose timers, watchdogs, and an interrupt controller. The MPU clock domain can be configured to run at four different frequencies: 350 MHz, 700 MHz, 920 MHz and 1.2 GHz. The frequency of both cores is scaled equally. The frequency of the memory subsystem is constant and independent of the MPU clock domain (200 MHz for the L3 Cache, 100 MHz for the L4 Cache and 400 MHz for the LP-DDR2).

The supply voltage of the MPU subsystem is provided by a Texas Instruments TPS62361 voltage regulator [144]. A low-pass filter is attached to this voltage regulator to suppress noise generated by the switching regulator. To measure the supply current of the CPU we inserted a shunt resistor after this low-pass filter (see Figure 4.7). The voltage dropping at this shunt was amplified using an INA199 shunt monitor [146] and then sampled using the 12bit analog digital converter of a MSP430 microcontroller [145]. The CPU's supply voltage at the output of the TPS62361 depends on the selected operating frequency of the processor and hence needs to be measured, too. As the low-pass filter is very sensitive to variations of the impedance, we first decoupled the filtered supply voltage using a voltage follower before connecting it to our measurement unit. Both, the voltage drop at the

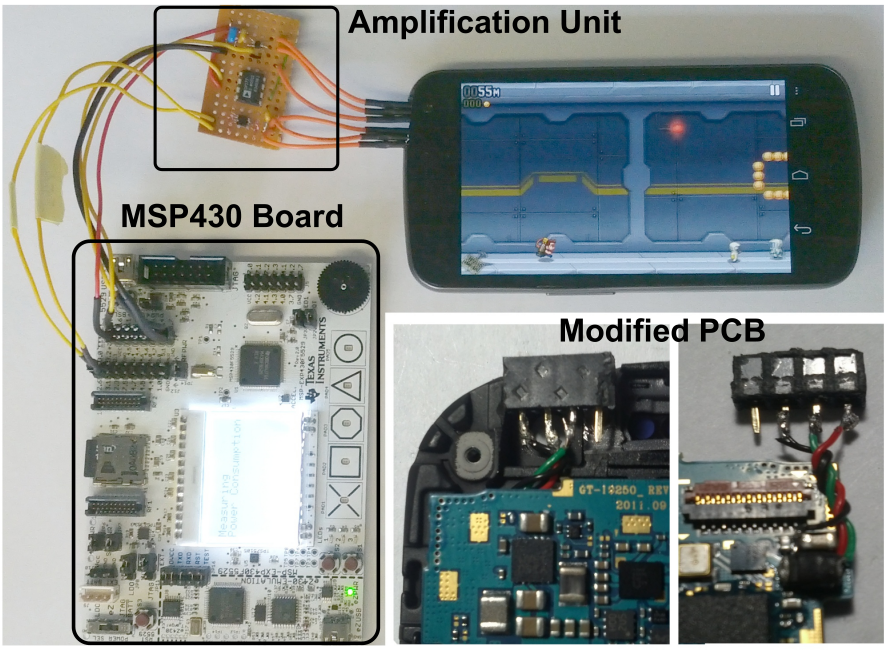


Figure 4.8: Experimental setup consisting of the modified Galaxy Nexus (modification see lower right), the amplification unit and the MSP430 measurement board

Table 4.2: Workload characteristics of the used games

Game	Avg. Workload $[\frac{cycles}{frame}]$	Deviation σ $[\frac{cycles}{frame}]$	Utilization [%] Min	Max
Cut the Rope	5.87e+06	6.89e+03	13	96
Jetpack Joyride	6.67e+06	8.92e+03	12	57
Temple Run	1.07e+07	9.55e+03	24	91

shunt and the CPU supply voltage were sampled at a rate of 20 kHz. The measurement unit was calibrated using a high resolution National Instruments measurement setup.

To allow reproducible measurements we ensured that the start and end of the measurement phase were synchronized with the start and end of the game. For this purpose, we removed the smartphone's front facing camera and attached the MSP430 to the corresponding I2C bus using a bi-directional voltage level-translator. This bi-directional communication between the phone and the measurement unit allowed us to perform synchronized measurements and to read the power measurements back to the phone which will be used in Section 4.5. Figure 4.8 illustrates the complete hardware setup.

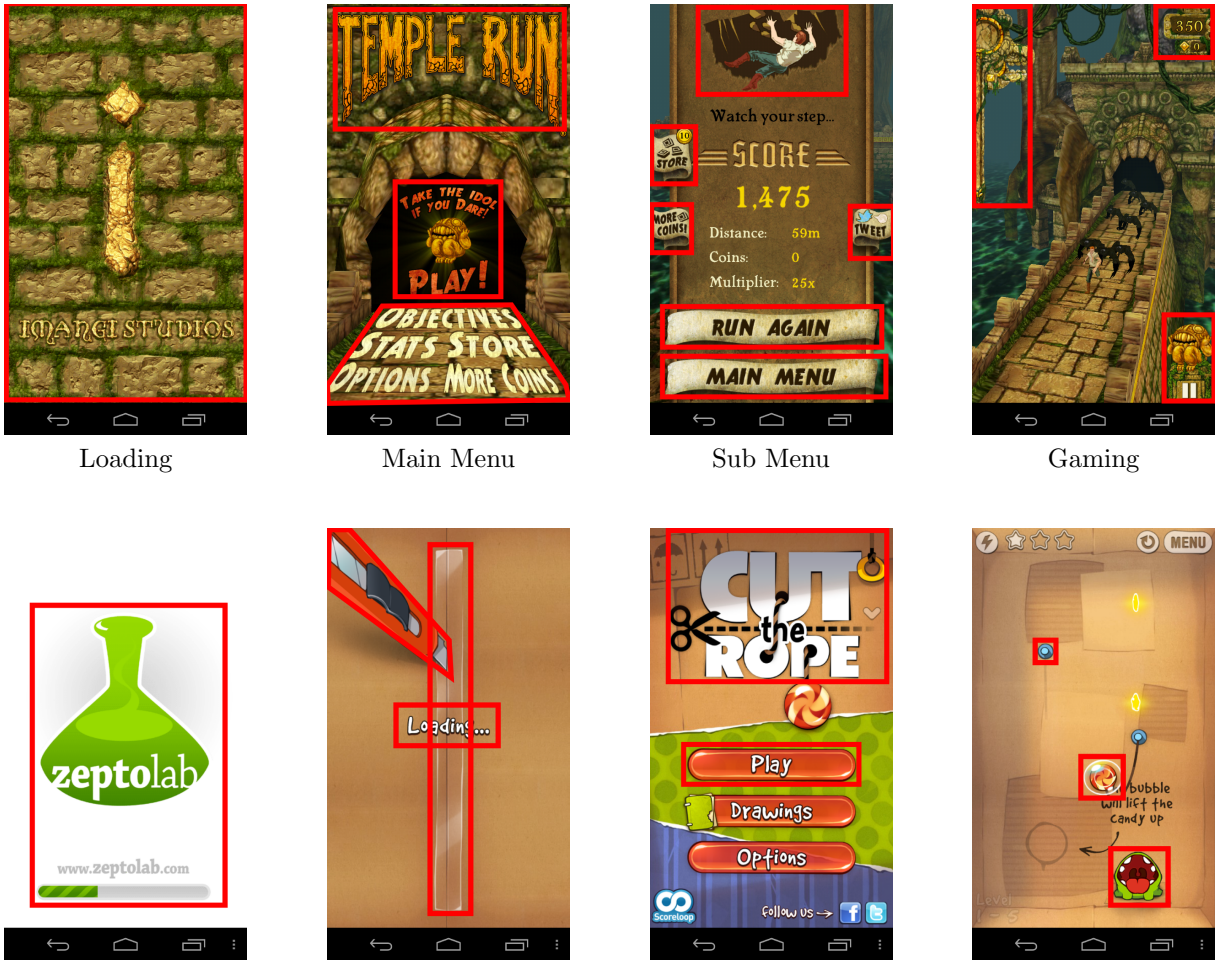
Selection of games: For our experiments we used three of the most popular Android games: ZeptoLab's Cut the Rope which is a puzzle game, the highly interactive endless running game Temple Run from Imangi Studios and the side-scrolling game Jetpack Joyride from Halfbrick Studios. The workload characteristics and the utilization of the

CPU at 1.2 GHz are given in Table 4.2. Clearly, Temple Run with its advanced 3D graphics imposes the highest workload on the CPU and shows the highest variation of the workload in terms of cycles per frame. The large utilization range shows that all of the games are amenable to DVFS.

Experimental methodology: When comparing the different power managers, it is of great importance to guarantee the repeatability of the experiments. None of the used games allowed to record and replay game plays as it is possible in Quake II. Moreover, the content of Temple Run and Jetpack Joyride changes from game run to game run. Hence, it was not possible to record and replay touch events using frameworks like Monkey Runner [9]. To still guarantee validity and significance of the experimental results, we performed the experiment as follows: We have put the phone into airplane mode to avoid distortions by pop-up advertisement and background synchronization services. For each measurement, the games were played for 5 minutes by the same player. The patterns of calling menus and selecting levels were kept constant if possible. For the game Cut the Rope, always the same levels have been played, i.e., Season 1, Level 1 to 18. Each experiment was repeated 5 times to guarantee that the results can be replicated and to capture possible variations. All the experimental results reported in the remainder were obtained following this scheme unless otherwise noted.

We have identified textures for all three games that allowed detecting the different game states. Figure 4.9 shows examples of such textures together with the screenshots of the corresponding states. A correct identification of the different states is of great importance as a wrong identification can either lead to a decreased gaming quality (e.g., if a gaming state is classified as a no interaction state and hence is only rendered with 20 frame per second) or to an increased power consumption (e.g., if for a menu state 58 frames per second instead of 30 are chosen as target frame rate). Since the games are all closed source and do not provide insight into their current state, we used visual inspection to verify the accuracy of the game state detection. For this purpose, the EGL library was modified to store all game frames and identified states on the SD card. This was done for all three games using game plays of 10 minutes each. A following visual inspecting allowed us to verify that the frames were classified correctly. For all the experiments performed, we did not observe any wrong classification of game states.

The game state detection allowed to perform a detailed workload analysis of the different states. Memory and CPU boundedness are of great interest when applying DVFS. If a game is highly memory bound, the CPU frequency can be reduced and hence power be saved without impacting the performance. If a game is CPU bound, the performance will be directly affected when the processing speed is reduced. A common way to determine the boundedness is to leverage the performance counters of the processor [25]. The processor of the Samsung Galaxy Nexus, however, is configured such that the CPU performance counters can not be read. To overcome this limitation, we have ran the games at all available processing frequencies and measured the frame rate as well as the time spent in the loading state. The results including the observed variations are presented in



Loading | Main Menu | Sub Menu | Gaming

Initial Loading | Level Loading | Main Menu | Gaming

Figure 4.9: Screenshot of different states of the games Temple Run and Cut the Rope. The red boxes mark regions that allow an unique state identification based on the underlying texture

Figure 4.10. It can be seen that for all games the high interaction state becomes CPU bound when reducing the frequency below 700 MHz. For all games, the low interaction state has higher performance requirements compared to the high interaction state. The loading time increases exponentially when decreasing the processing frequency. In the case of Jetpack Joyride, it can be seen that increasing the frequency from 920 MHz to 1.2 GHz resulted in a reduction of the frame rate. This is due to the internal overheat protection of the OMAP4460 which forced the processor during the game plays to scale down to 350 MHz several times.

In the following we present the results based on the described setup and obtained with both, the `generic game` and the `game state specific` governor.

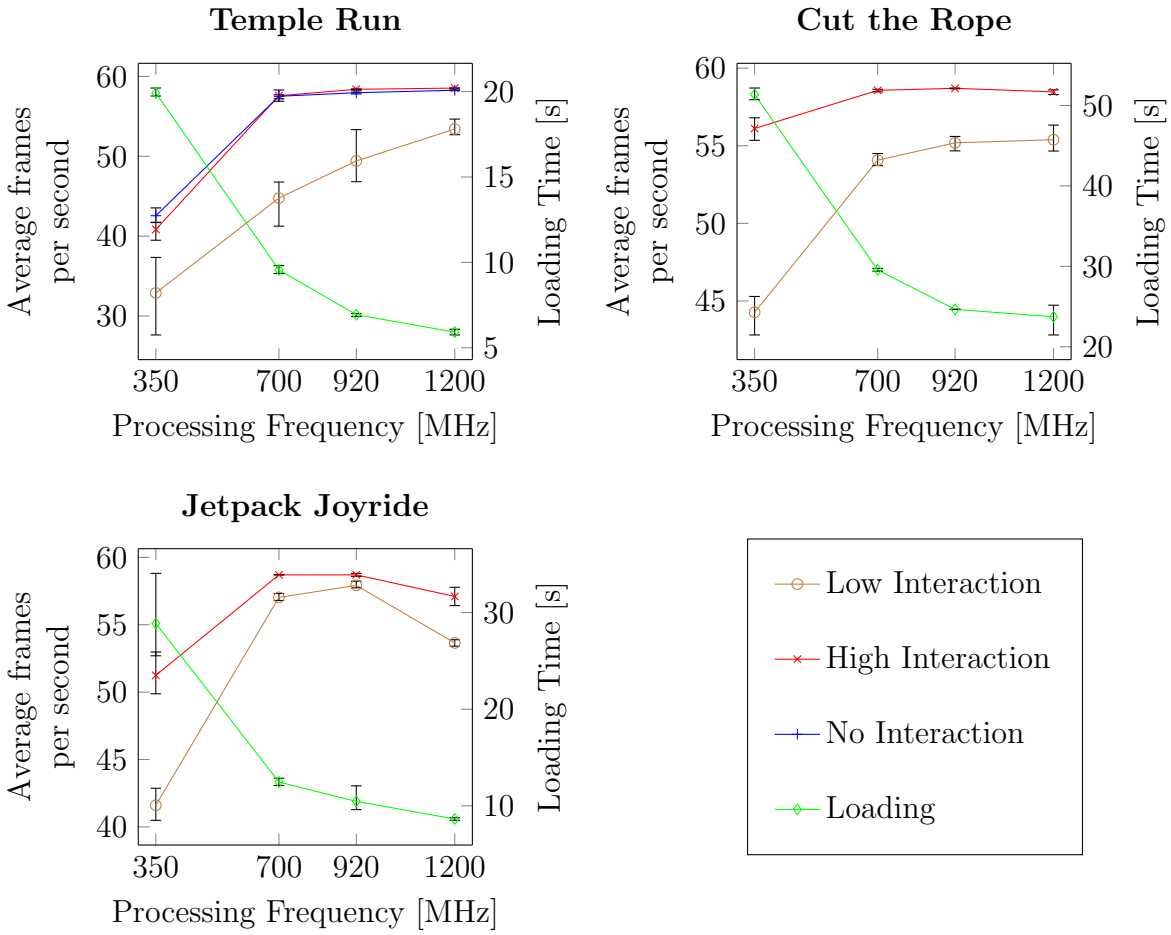


Figure 4.10: Loading times and frame rates for the available processing frequencies

4.5 Experimental results

We have modified Android according to Section 4.1 and implemented both governors described in Section 4.2 and 4.3. Each DVFS power management scheme includes an overhead in terms of required computations, instrumentation overhead and switching overhead which will be presented first. By presenting a detailed quantitative evaluation of the different overheads, we allow a direct comparison to other existing power management techniques.

4.5.1 Overhead measurements

After each frame, the kernel is notified of the swap event by the game power management library (see Figure 4.2). The overhead for this notification was measured to be 989.7 cycles per `ioctl` in average. This overhead is considered as negligible compared to the computation time of a frame (see Table 4.2).

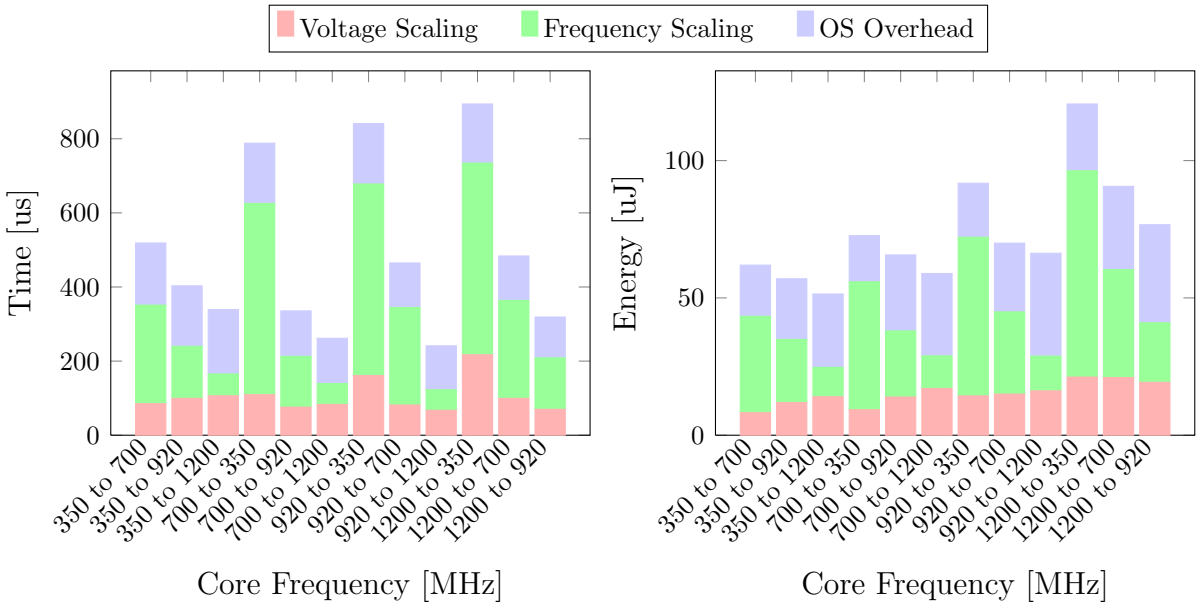


Figure 4.11: Frequency scaling overhead in terms of time and energy measured for the TI - OMAP4460

The prediction given by Equation (3.6) has been implemented using fixed point arithmetic as there was no floating point unit (FPU) support in the kernel by the compiler used. One prediction of the order $n = 10$ consumed in average 207.4 cycles which is considered as completely negligible compared to the processing time of a frame.

Switching the voltage and frequency of a processor always involves costs in terms of energy and time. Figure 4.11 gives the overhead in terms of time and energy it takes to switch from the current processing frequency to a particular target frequency. This overhead includes the whole pre-scaling notification process implemented in the Linux kernel, the time consumed by the scaling of the voltage, the settle time of the PLL, updating the jiffy counter, as well as the post-scaling notification. To measure these time and energy overheads, the `cpufreq` Kernel driver has been instrumented to toggle general-purpose I/O (GPIO) pins of the phone at entry and exit points of related functions. The voltage at the shunt (see Section 5.2) and the logic states of the GPIO Pins were sampled using a National Instruments PXI-6124 card and a sampling rate of 2 MS/s. The frequency transitions were initiated using the `userspace` governor. The measurements were repeated 10 times for each frequency step to avoid any imprecisions. As one can tell from Figure 4.11, the average switching overhead highly depends on the current and the target frequency. Switching directly to the lowest processing speed of 350 MHz is most expensive with up to 893.9 μs. With a target frame rate of 58 frames per second resulting in a deadline of 17.24 ms this switching duration cannot be neglected. Hence, the governor has been programmed in a way that it takes the overhead into account when making the decision about the next frequency. Further, we have implemented a *lazy* switching algorithm which delays the frequency switches to the lowest frequency by one

Table 4.3: Overhead caused by the computation of hash keys over the texture data for all occurring `glTexImage2D` calls

Game	Number of textures	Data [kbyte]	Time [s]	Overhead [%]
Jetpack Joyride	448	75601	0.45	5.93
Temple Run	45	13326	0.12	1.85
Cut the rope	94	84481	0.50	5.75

frame and switches only if the lowest frequency is requested again. A comparison between the `generic game` governor with and without the *lazy* switching enabled is presented in Section 4.5.3.

For the `game state specific` governor we have to additionally compute a hash key for each texture being loaded. All of the used games make use of compressed and uncompressed textures which are loaded using different OpenGL calls. We observed that instrumenting only the OpenGL call that is responsible for loading uncompressed textures, i.e., `glTexImage2D`, is already sufficient for a robust state detection. For the used games, nearly all `glTexImage2D` calls were made during the loading phase. Table 4.3 reports the number of calls made in total, the amount of data over which hashes had to be computed, the computation time and the overhead in ratio to the loading time without the hash computation. The total loading time of the game was prolonged by 5.93 % in the worst case. Considering the achieved energy savings reported in the following this overhead is considered as tolerable. Note that this overhead could be completely avoided if Android provided an API allowing the game developer to directly inform the operating system about state changes.

4.5.2 Critical speed

The described `generic game` and `game state specific` governors are allowed to choose among all available processing frequencies of the CPU. Slowing down the CPU, however, might prolong computations and the time other smartphone components, like the GPU or the memory, are active. Hence, decreasing the CPU frequency and power consumption, might lead to an increase of the smartphone's overall average power consumption. As described in Chapter 2, the critical speed is defined as the processing frequency that minimizes the system's overall energy consumption for a given task. We have measured the mobile phone's total average power consumption for all available CPU processing frequencies to confirm that in our case the critical speed is actually the lowest available processing frequency. Figure 4.12 clearly shows that the phone's average power consumption continuously decreases when lowering the CPU processing frequency. Hence, it can be concluded that the critical speed for our setup and type of applications is the minimum processing speed of the CPU, enabling the governors to choose among all available processing frequencies.

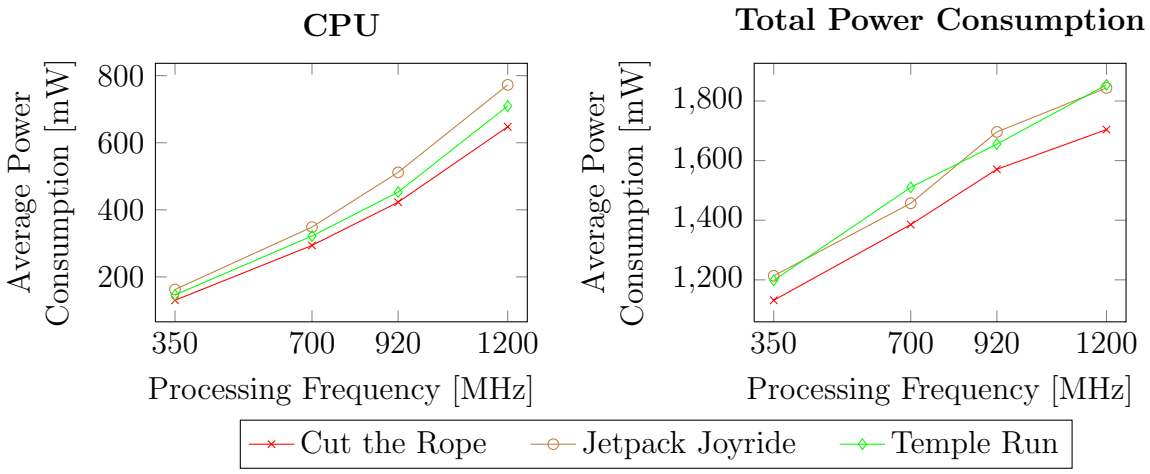


Figure 4.12: Average CPU and smartphone's total power consumption for all available processing frequencies

4.5.3 Generic game power management results

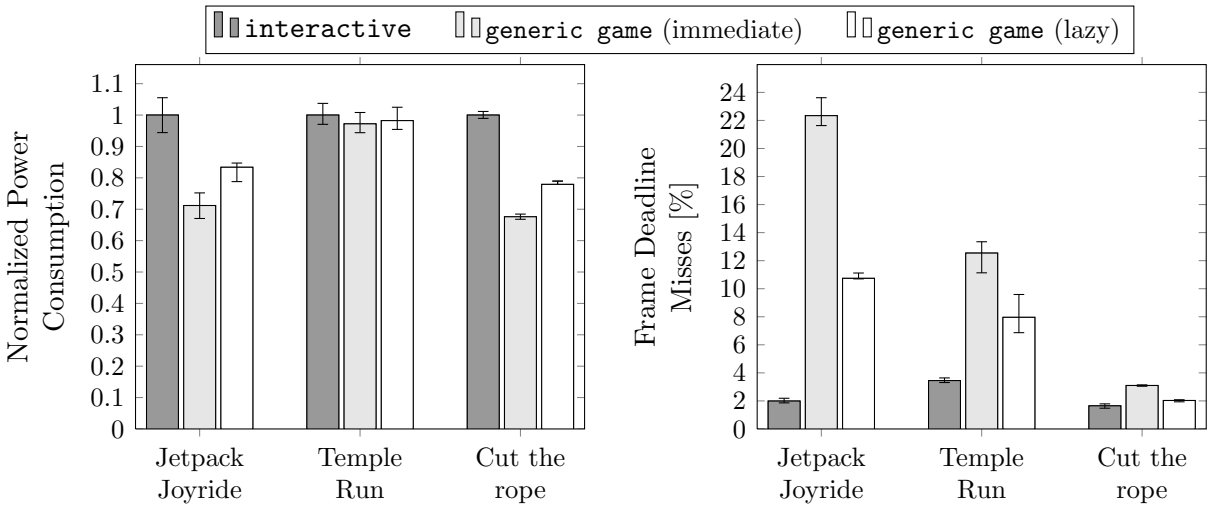


Figure 4.13: Comparison of the average power consumption and percentage of frame deadline misses between `interactive` and the `generic` game governor (immediate and lazy switching)

We have tuned the weights of the auto-regressive model based predictor offline for each game individually like described in Section 4.2. At the beginning of a game the `generic game` governor is notified about the current game's name and automatically chooses the corresponding weights. Figure 4.13 depicts the average power measurement results including the observed variations obtained for the three different games described in Section 4.4. Clearly, it can be seen that if using the `generic game` governor in combination with *immediate* switching the average power consumption can be reduced significantly compared

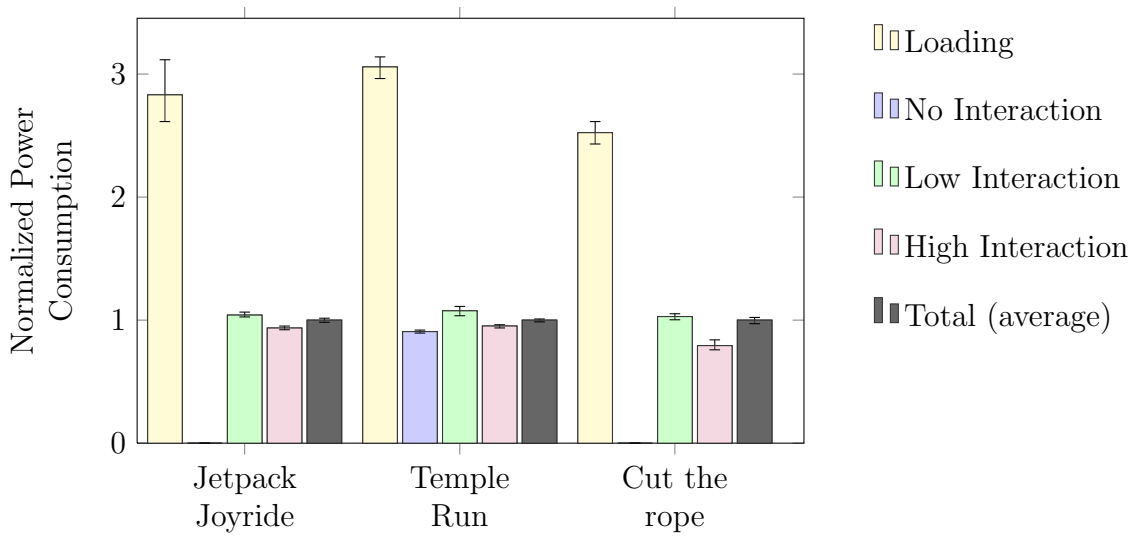


Figure 4.14: Breakdown of state power consumption in ratio to total power consumption using the `interactive` governor

to the `interactive` governor. In case of Cut the Rope a reduction by 32.4 % of the consumed power was possible whereas the percentage of frames missing their deadline was only increased by 1.45 %. Similar savings could be achieved for Jetpack Joyride (28.8 %). In the case of Temple Run only 2.8 % could be saved. For Jetpack Joyride and Temple Run the amount of frames that missed their deadline was 22.3 % and 12.5 % respectively. This percentage, however, could be be drastically reduced to 10.7 % and 7.9 % respectively by using the *lazy* switching algorithm while only increasing the average power consumption by 1 % for Temple Run and 12.2 % for Jetpack Joyride. With the `generic game` governor and *lazy* switching enabled no difference in terms of playing quality could be perceived for any of the three games as will be shown in Section 4.5.5. Hence, all results presented in the following use the *lazy* switching algorithm. In Section 4.5.5, we investigate if these percentages of frame deadline misses have any impact on the by the player perceived quality.

All used games have different states (like the loading and menu state) which might require different power management strategies. In the following, we will first detail the state specific power consumptions and show how the above presented power savings can be even further increased using the `game state specific` governor.

4.5.4 State-specific power management

We have incorporated the state detection described in Section 4.3 into the OpenGL library, identified prominent textures for all games and states as described in Section 4.4 and thereby generated the `<hash,state>` map required for the detection algorithm. This, in turn, allowed us a detailed analysis of the power consumption of each individual game state. For each frame we requested the accumulated power consumption and the num-

Table 4.4: Analysis of processor's energy consumption during the initial loading phase of Cut the Rope for fixed frequencies, the `interactive` and the `generic` governor

Frequency	Time [s]	Power [mW]	Energy [mJ]
350 MHz	21.9	131.4	2877.7
700 MHz	11.5	355.6	4089.4
920 MHz	10.3	557.5	5742.3
1200 MHz	8.7	896.4	7798.7
`interactive`	8.8	781.0	6716.6
`generic`	9.6	668.7	6419.5

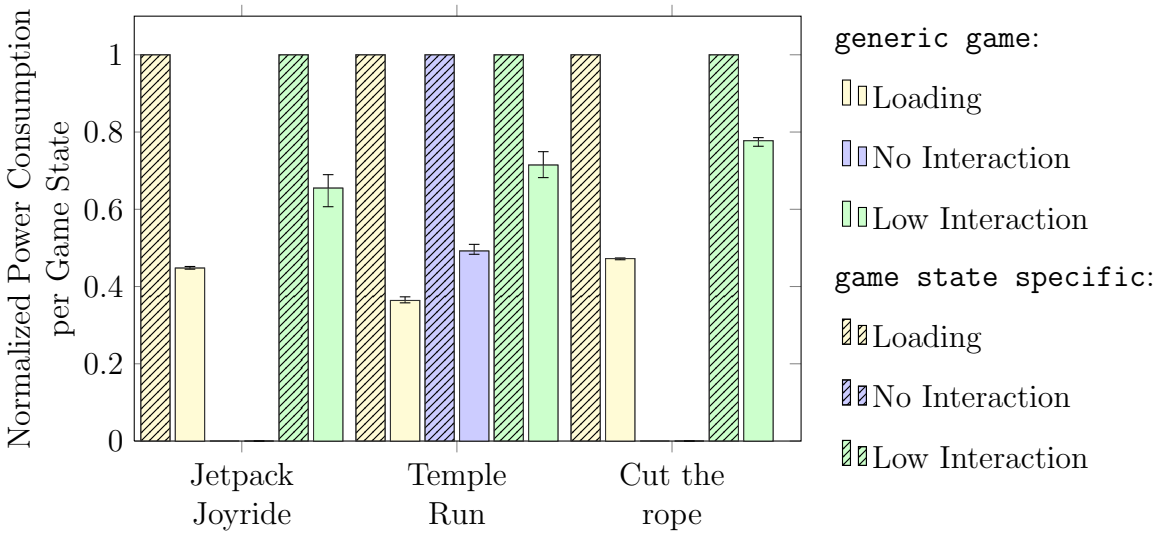


Figure 4.15: Comparison of the `generic` and the `game state specific` governor for each state individually

ber of samples taken from the MSP430 measurement unit. These samples were again accumulated for each game state individually. Once the game was exited we computed the average power consumption for each state. Figure 4.14 depicts a breakdown of the average power consumption in ratio to the total average power consumption and including the variations for the different states and games using the `interactive` governor. It can be seen that the CPU consumes a considerable amount of power during the loading state. For all games the low interaction phase consumed more power than the playing phase itself. The largest variation between the individual experiments was observed for the loading state. For the other states nearly no variation could be observed. In the following, we are going to detail the individual power measurement results using the `game state specific` governor.

Loading: As described in Section 4.3, during this state mostly data is loaded into the main memory and objects are initialized. This phase is very likely to be memory bound which means that the CPU spends a lot of time on waiting for the memory. As a con-

sequence, the default `interactive` governor will detect a high system utilization and therefore is likely to use high processing frequencies. The `generic game` governor will detect a low frame rate and will as well use high frequencies to increase the frame rate to the target frame rate set. In order to determine the optimal frequency to be used during the loading state, we measured the loading time and the average power consumption during the initial loading phase for all available processing frequencies. Table 4.4 gives the energy consumed during the initial loading phase of Cut the Rope for the different fixed processing frequencies, the `interactive` and the `generic` governor. The presented measurement results include only the initial loading phase during which always the same content is loaded. By this we avoid variations in terms of loading time and consumed power due to varying content like level specific data. Hence, the power measurement results in Table 4.4 differ from the results shown in Figure 4.10 and 4.14.

It can be seen that reducing the processing speed from 1200 MHz to 700 MHz increases the loading time only by 2.8 seconds, but reduces the amount of consumed energy by 47.6 %. If the frequency gets reduced further to 350 MHz, the loading state becomes CPU bound and the loading time is significantly increased by 13.2 seconds compared to the shortest possible loading time. The same behavior was observed for the other two games: For Jetpack Joyride the energy consumed during the loading phase was reduced by 23.8 % (if the processing speed is lowered from highest to 920 MHz) while the loading time was only increased by 1.9 seconds. With an even more aggressive power manager it was possible to reduce the energy consumption by 41.9 % (using 700 Mhz) if a loading time increase from 8.5 to 12.4 seconds is considered as tolerable. For Temple Run the loading energy could be reduced by 30.8 % and 43.4 % while the loading time increased from 5.9 seconds to 6.9 and 9.5 seconds respectively. For all three games reducing the frequency to 350 MHz resulted in the best energy savings, but increased the loading time by an intolerable amount of time. Hence, for the three games using a processing constant frequency of 700 MHz was considered as optimal solution for the loading state and is therefore used by our `game state specific` governor. Compared to the default `interactive` governor, this resulted in power savings of 39.1 % while the loading time was only increased by 2.7 seconds in the case of Cut the Rope. A possible extension of the manual tuning could be to take the processor's performance counters into consideration on smartphones where this is possible and thereby automatically determine the optimal frequency.

No Interaction: The game Temple Run has a non-interactive movie scene directly after the initial loading state. As shown in Figure 4.15, a reduction of the target frame rate to 20 frames per second during this state led to power savings of 50.8 % in case of Temple Run and compared to the `generic game` governor. The other two games used for this study have no movie scenes and therefore no results can be presented.

Low Interaction: This includes all menu scenes where the user can interact, but typically in a slower fashion. Again, we reduced the target frame rate, this time to 30 frames per second as this did not impact the perceived quality for any of the games. In case of Temple Run the average power was reduced by 28.6 %, for Jetpack Joyride by 34.5 % and for Cut the Rope by 22.3 %.

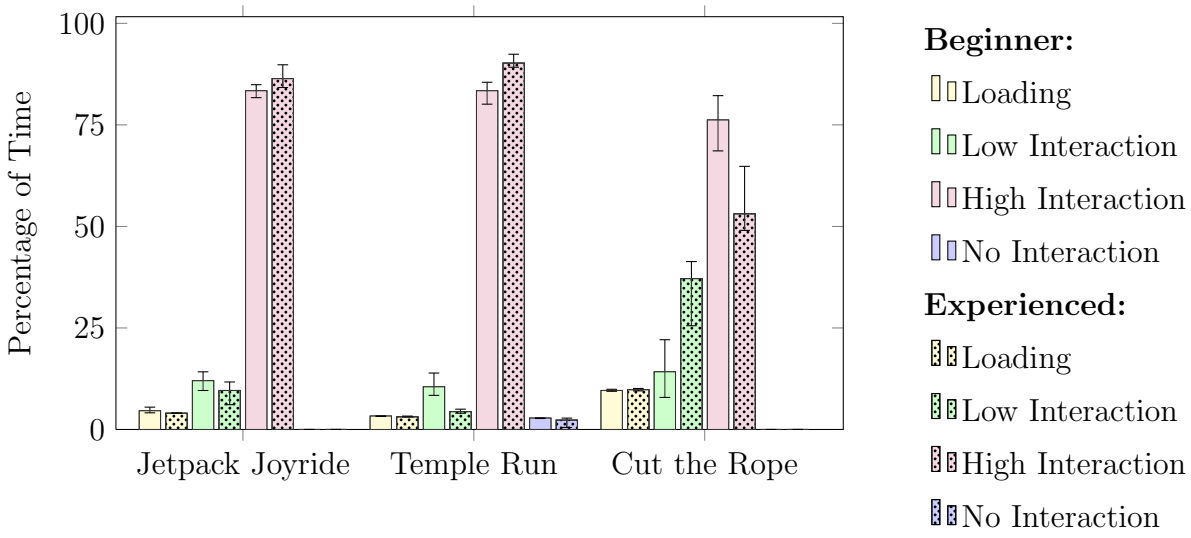


Figure 4.16: Percentage of time spent in the different states by experienced players and beginners

High Interaction: The target frame rate for this state was set to the monitor refresh rate of 58 Hz to guarantee maximum gaming quality. As this is the same as used by the `generic` governor during all states, the `game state specific` governor consumes the same amount of power during this state. The results are therefore omitted in Figure 4.15.

It has been shown that the power consumption can be significantly reduced for each state individually. The overall power savings, however, will as well depend on how much time the player spends in the different game states. Time, again highly depends on the playing skills and the type of game as shown in the following. To evaluate this impact, we have performed the following experiment: For each game, we asked 4 beginners and 4 experienced players to play the game for 3 times 5 minutes. Beginners, in contrast to experienced players, have never played the game before. The players were not aware of the intention of our measurements. We recorded the time the players spent in the different states. Figure 4.16 shows the average percentage of time the players spent in the corresponding states during the 3 runs, including the variation. It can be seen that especially for the game Cut the Rope, the level of experience highly influences the amount of time spent in the menu and gaming states. As described in Section 4.4, Cut the Rope is a puzzle game. After a puzzle is solved, the game enters a low interaction state where the score is shown and the player has to press continue to start the next puzzle. Experienced players were able to solve the puzzles quickly and hence spent a significant amount of time in the menu state (37.2 % in average). Beginners, on the other hand, spent only 14.2 % of the total time in the menu state and 76.2 % in the gaming state as it took them longer to solve the puzzles. In contrast, for the endless running games Temple Run and Jetpack Joyride, skilled players were able to run longer and hence spent less time in the menu state. In the case of Temple Run, experienced players only spent 4.4 % of the total time in the menu state whereas beginners spent in average 10.5 % in this state. The loading

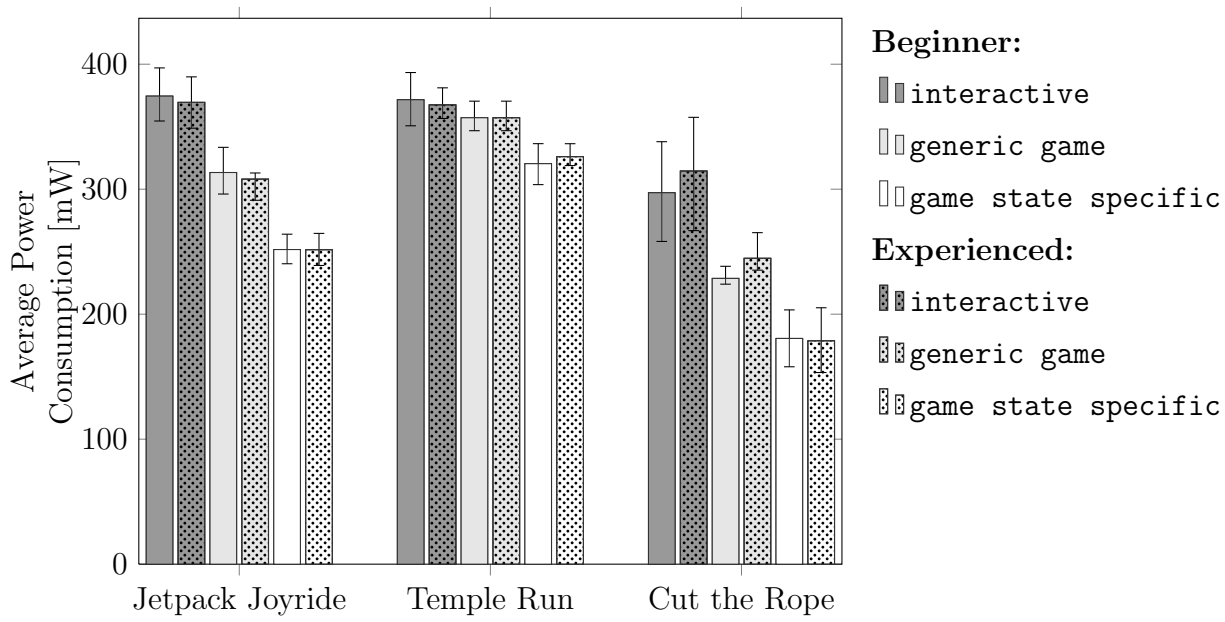


Figure 4.17: Comparison of the average power consumption for the different governors and different type of players

time was identical for all players. The time that was spent in the no interaction state of Temple Run differed between less than 1 second to 8.2 seconds since some players were aware of the possibility to skip the movie scene.

The implications of these results on the possible power savings are visualized in Figure 4.17. The `game state specific` governor reduced the average power consumption compared to Android's `interactive` governor in average by 43.2 % for the game Cut the Rope, by 31.9 % for Jetpack Joyride and by 13.8 % for Temple Run. For all games, the average power consumption differed between beginners and experienced players only to a small extent (up to 4 % for Temple Run). However, the variation intervals indicated in Figure 4.17 clearly show that the amount of time a player spends in the different states will have a large influence on the power consumption. For the game Cut the Rope, variations of up to 25.3 % could be observed.

Besides the time spent in the different states, the way the player interacts with the game might as well directly impact the power consumption. For example, an aggressive player might cause more input events, in particular during the gaming state, than a relaxed player. If such variations affect the power consumption depends completely on the way the game processes, e.g., input events. When comparing the power consumption of all 8 players to each other (for each game individually), the maximum variation occurred during the gaming state of Jetpack Joyride with 17.2 %. In comparison, the maximum variation observed when comparing only the 3 different runs of each player was 6.5 % for Jetpack Joyride. On the contrary, during the gaming state of Temple Run the maximum variation among all 8 players was only 5.4 % while the variation between the runs of the single players was 3.6 %.

From the measurement results presented in this section we can conclude that significant power savings can be obtained if the `game state specific` governor is used instead of Android's default `interactive` governor. However, as it has been shown in Figure 4.13, using the `game state specific` governor results in a higher percentage of frame deadline misses. We have performed a user study to answer the question if these frame deadline misses impact the by the user perceived quality.

4.5.5 User study

Table 4.5: Results of the user study

Game	`interactive`			`state specific`			t-value	p-value
	$\overline{x}$	σ_x	$SE_{\overline{x}}$	$\overline{x}$	σ_x	$SE_{\overline{x}}$		
Cut the Rope	4.92	0.28	0.08	4.92	0.28	0.08	0.00	1.00
Jetpack Joyride	4.85	0.38	0.10	4.62	0.87	0.24	0.87	0.37
Temple Run	4.85	0.38	0.10	4.69	0.48	0.13	0.91	0.40

In the work presented in [38,39], it has been assumed that users don't notice any difference between Android's default governor and the developed approaches. We have performed a user study to test if this assumption holds true. The study is based on three games, namely Cut the Rope, Jetpack Joyride and Temple Run. We asked 13 users to play each game for 3 minutes with two different settings: Once with Android's default `interactive` and once with the `game state specific` governor. After each game play the users were asked to rate the game quality of the current setting on a range from 1 to 5. We randomly selected the order in which governors were presented to the users to avoid any influences on the results. In total each user played games for approximately 18 minutes (6 game plays of 3 minutes each). Based on the user ratings, Table 4.5 was derived which shows the average rating $\overline{x}$, the standard deviation σ_x, the standard error $SE_{\overline{x}}$, the t-value and the p-value for all three games.

It is hypothesized that

H_0: *Users do not notice any difference between the* `interactive` *and the* `game state specific` *governor.*

From Table 4.5 it can be seen that the `interactive` governor in average has been rated better in case of the games Jetpack Joyride and Temple Run (see Table 4.5). We have performed the one-tailed t-test to analyze if the difference between the two means is statistically significant or not, i.e., if the difference can be contributed to a real difference in terms of quality between the two governors or to natural variations due to the low number of participants. From the small t-values of 0.0, 0.87 and 0.91 determined for the

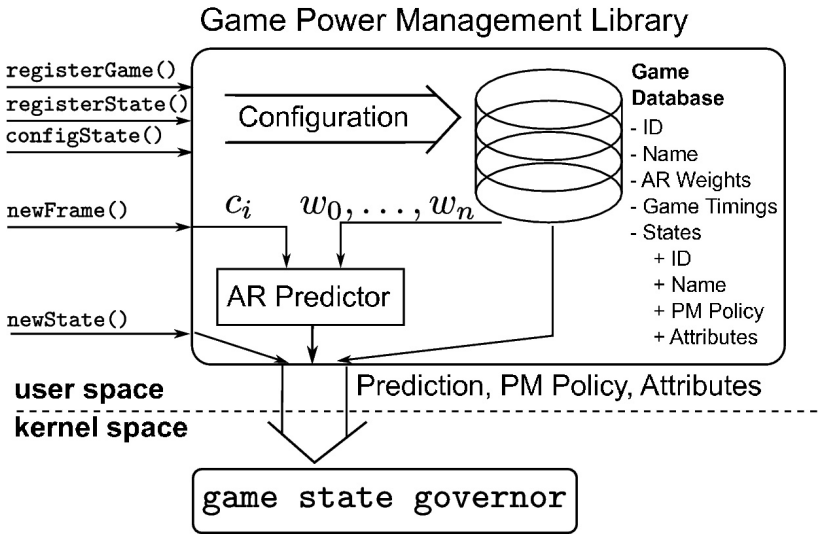


Figure 4.18: Game power management API

three games (Cut the Rope, Jetpack Joyride and Temple Run) one can conclude that the null-hypothesis cannot be rejected with a confidence of 95 %. Further, the high p-values of 1.00, 0.37 and 0.40 support this fact and strongly indicate that the observed difference can very likely be contributed to natural variations. This in turn indicates that both governors perform equally well in terms of provided user experience. As detailed in the previous section, using the `game state specific` governor results in a significantly lower power consumption and hence should be preferred over Android's default governor.

4.6 Game power management API

We have shown how the game's timing information and the knowledge about the game's current state can be leveraged to significantly reduce the power consumption. The information was obtained by intercepting the game's communication interface to the underlying OS. This data, however, could be directly provided by the game developer if there was an API provided by Android. Thereby, the overhead described in Section 4.5.1 could be significantly reduced and the identification of state-specific textures for each game would no longer be required.

A possible layout of such an API is given in Figure 4.18. Before the game power management can be used, the developer needs to register the game with the power management module. This will either create a new entry in the game database or load existing settings. Additionally, all of the game's states together with the desired power management policies need to be configured. In this work, we have applied two different power management policies: The first policy targets to maintain a state-specific frame rate. This target frame rate should be provided by the game developer for all existing states. The second policy optimizes the power consumption during memory bound phases. To determine the

optimal frequency for this state, the performance counters of the CPU can be leveraged as reported in [25]. Besides configuring the game power management library, the game developer needs only to instrument the game's source code to report state changes and the beginning of new frames. As described in Section 4.2, a possible choice for predicting the next frame's workload are auto-regressive models. Since such models require the model parameters to be tuned, the API should provide the possibility to record the frame timing information and perform the tuning, e.g., based on Marple's least squares algorithm [100]. Once the weights are determined, the game power management library can forward the prediction results, the current policy and attributes like the target frame rate to the `game state specific` governor which performs the scaling of the frequency.

4.7 Summary

This has proposed a technique to detect a game's current state based on its communication with the underlying operating system. The state-specific power management scheme uses the knowledge of the game's current state to significantly reduce the power consumption of the processor. The savings are achieved by scaling the voltage and the frequency of the processor dynamically depending on the future workload of frames. We showed how the timing information, which is required by the workload predictor, can be gathered by the proposed graphics library instrumentation. The technique does not require the game's source code and is not specific to a game, nor the operating system, nor the processor. By this we achieve great portability and open up the possibility for the research community to experiment with different games, game states and workload prediction techniques.

To prove the efficiency of the concept, we implemented this technique on an Android-based Samsung Galaxy Nexus smartphone. We modified the phone to allow detailed measurements of the CPU's power consumption. Based on this setup, we for the first time gave insight into the processor's power consumption during the different states of a game and revealed that not only during the highly interactive gaming state, but as well during the loading and menu states a significant amount of power is consumed. In the experimental section, we showed significant power savings: Compared to the default `interactive` governor coming with Android we were able to reduce the power consumption by up to 43.2 % using our `game state specific` governor.

We have confirmed, based on a user study, that our governors do not impact the gaming experience. Towards this, we have asked users to rate both, Android's `interactive` and our `game state specific` governor. No significant statistical difference could be observed between the average ratings of the governors.

The results presented in this chapter point out the need for a game power management API on mobile devices. Such an API would allow game developers to provide information to the operating system, which is currently only available inside the game: (i) the game's current state, (ii) timing measurements and (iii) the desired frame rate for the current state. Moreover, the game developer has in many cases insight in future workloads that

might arise due to particular game events. Based on that, it might even be possible for the game developer to provide an accurate workload prediction to the power manager and by that highly increase the prediction accuracy. In addition to the power savings that might be possible based on the provided information, such an API would completely remove the current overhead imposed by the state detection algorithm.

We have shown how to intercept the communication of the game with the graphics library OpenGL to gain information about the game's current state and the frame timing information. There are several other interfaces like the audio library OpenAL, system calls issued by the game and touch events reported to the game that might be leveraged in future to perform an even finer-grained power management. This work focuses on reducing the power consumption of the CPU. However, the presented game state detection might as well be used to perform game state-specific power management for other components such as the GPU, the display and sensors like accelerometers.

We have shown how to further improve the power manager, presented in Chapter 3 by introducing an awareness of a game's current state. One important question, however, has not been answered, yet: How much power savings are theoretically possible if the power manager exactly knew future requirements of a game and therefore could optimally scale the voltage and frequency of the processor. In the next chapter, we describe a model that allows answering this question and thereby motivate continuing research in the domain of game power management.

5

Estimating power management limits

In the previous chapters, we have shown how the CPU's power consumption can be considerably reduced by up to 43.2 % compared to Android's default power manager. Towards this, we have evaluated different workload predictors in respect to their suitability to forecast game workloads. We have shown that the autoregressive model-based predictor, the most general form of linear models, provides a good performance in terms of power savings and does not affect the by the user perceived quality. Further, we have developed techniques to identify different game states and perform state-specific power management. While AR model-based prediction significantly outperformed Android's default governor, it is not clear if more complex workload predictors, e.g., non-linear models, provided a significantly better performance or not. In this chapter we present a method to accurately approximate a perfect game workload predictor, in the following referred to as *oracle* predictor, and its resulting power consumption. Thereby, we reveal the remaining gap between *current* and *optimal* techniques and motivate future research endeavors.

The **contributions** of the work presented in this chapter are as follows:

- We present a technique that allows to accurately model the optimal game power manager. Towards this, we have investigated two different approaches. First, we consider a frame-based model which simulates the behavior of the CPU at the different processing frequencies for each individual frame. We disclose that the non-deterministic nature of games and the high inter-dependency of computations with, e.g., GPU, memory and I/O, does not allow such frame-accurate modeling. Hence, we introduce a statistical model, which does not allow deriving the optimal

performance of each individual game frame, but accurately provides the performance in terms of power and percentage of frame deadline misses for a full game play.

- We introduce a power model that allows approximating the power consumption of the CPU based on the extent to which each individual frequency is used and the average power consumption of frames at a particular frequency. We validate the model observing a maximum deviation of 6 % from the real measured power consumption. Based on the statistical and the power model, we present results of the optimal game power manager. Even though the AR-based governor outperforms Android's default governor, we show that there is still a significant gap to the performance of the optimal power manager. For the game Temple Run we for example observed up to 54.37 % of additionally possible power savings compared to the AR-model based approach.

- Ultimately, the real limits of game power management depend on the individual game, the by the user desired number of frames per second during the individual states and the tolerable frames missing the deadline. These factors can highly vary between users and games. To evaluate these differences and provide concrete numbers a detailed user study has to be performed which is not the scope of this work. In this work we provide the full picture by presenting purely quantitative results for three games, each from a different genre, the full range of viable frame rates ranging from 20 to 50 frames per second and for the individual game states.

- Our detailed analysis reveals reasons for the remaining gap, provides future directions and strongly motivates future research endeavors towards power efficient gaming. We illustrate possible approaches that might further increase the performance of current power managers.

- Race-to-halt is a popular alternative to DVFS. In case of gaming applications, the frequency is set to the maximum at the beginning of the frame and, as soon as the processing is completed, the CPU is put into a sleep mode until the beginning of the next frame. We show that race-to-halt consumes significantly more power than the AR-based governor and consequently cannot be considered as an alternative.

Related work: Optimal power manager models have been widely discussed in literature. Especially in the real-time domain, schedules are derived that optimize the performance and power consumption of processors under given real-time constraints [11, 12, 117, 119, 159]. Models of optimal power managers have as well been proposed for multimedia applications. In [16], the performance of the minimum possible energy consumption of a video decoder is derived by formulating the optimal DVFS as a linear program. Pering et al. [118] simulate the performance of different power managers for video decoding applications and compare them to the theoretical optimum. An approximation of the optimum is obtained by a brute-force approach, i.e., traces are generated for all frequencies and, based on post-simulation analysis, the optimal voltage and frequency settings are determined. As will be explained in Section 5.1, this approach cannot be applied to game workloads due to their non-deterministic nature.

All of above work is based on simple task models, like purely periodic tasks, or simplified processor models, e.g., assuming linear scaling of the processing time with the frequency. Inter-dependencies of CPU computations with other peripheral components like the GPU, memory or I/O are not considered. This makes proposed models inappropriate to model the optimal game power manager.

Organization of the chapter: Section 5.1 describes two different approaches to accurately model the optimal power manager: A frame-based and a statistical approach. We show why a frame-based approach is not applicable for games and detail the statistical model. Section 5.2 describes the hard- and software setup used for this study. In Section 5.3 we present the experimental results and compare the performance of existing techniques to the optimal power manager. Section 5.4 concludes this chapter and highlights the implications of the results.

5.1 Optimal power manager model

Games are, similar to videos, frame based. To optimize the power consumption, for each frame i the minimum possible frequency $f[i]$ should be used that just avoids violating the deadline $1/FPS$. To determine the theoretical limits of power management, we assume an *oracle* predictor, i.e., a purely theoretical predictor that exactly knows the future and hence can choose the optimal frequency for each frame. This predictor provides the optimal sequence of frequencies $S = (f[0],\ f[1],\ \ldots,\ f[n])$ that minimizes the power consumption. In the following we are going to discuss the challenges of a frame-based model that accurately provides this optimal sequence.

5.1.1 Frame-based model

In the case of videos the frame content and the corresponding workload for de/encoding is deterministic. Hence, each frame can be simply de/encoded on the real hardware using all available frequencies and thereby the optimal sequence of frequencies can be determined for all frames of the clip [118]. As described in Section 3.3.3, for games, on the contrary, the content depends on the user actions as well as the time it takes to process each frame: At the beginning of each frame a game typically computes the time Δt that has passed since the last frame. Δt is then used to update object positions, the physics engine and the artificial intelligence (AI). Hence, what exactly will be computed during the next frame, heavily depends on Δt. If a game is now played at different frequencies the processing time of individual frames, Δt and consequently the game content will completely differ from the previous run. This prevents a brute-force solution that is viable in case of video de/encoding.

Starting from Android 4.1, Project Butter [57] introduced the synchronization between the processing of frames and the vertical synchronization signal (VSYNC) of the display. As depicted in Figure 5.1, the processing of the next frame i always starts with the arrival

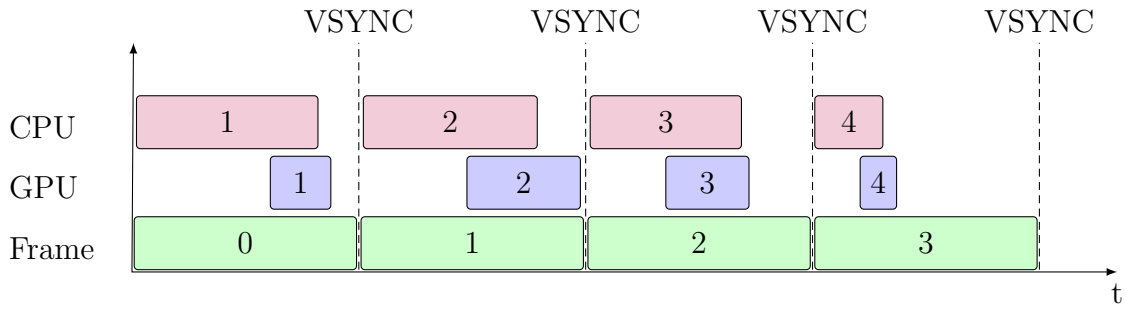


Figure 5.1: Ideal frame timing (all frames are processed in time)

Table 5.1: Average frame duration at different processing frequencies and workloads in ratio to the average frame processing time at the highest processing frequency (1200 MHz)

Workload	350 MHz	700 MHz	920 MHz	1.2 GHz
Temple Run	2.25	1.31	1.08	1.00
Cut the Rope	1.77	1.26	1.14	1.00
Jetpack Joyride	2.69	1.53	1.01	1.00
Purely CPU-bound	3.43	1.71	1.30	1.00

of a VSYNC signal. Due to this synchronization, in the ideal case the above described Δt will always equal to $1/r_{VSYNC}$, where r_{VSYNC} is the refresh rate of the display (typically 60 Hz). Based on this observation, we investigated the possibility to derive the optimal sequence from a single, fixed frequency recording of a game play. We instrumented the operating system (see Section 5.2) and recorded the number of cycles $c[i]$ required for each frame of a game play at the highest processing frequency. A simplified assumption is that the processing time linearly scales with the CPU's processing frequency. Hence, the optimal sequence of processing frequencies can be chosen by selecting the smallest frequency $f[i] \in \mathcal{F}$ for each frame, where $\mathcal{F}$ is the set of all available frequencies, that guarantees

$$\frac{c[i]}{f[i]} < \frac{1}{r_{VSYNC}}. \tag{5.1}$$

Assuming, that we never violate this deadline it should be possible to determine the optimal frequency for each frame. As described in Section 2.3.2, the linear relationship in Equation 5.1 typically does not hold true and in most cases is highly pessimistic. The frame computation time is amongst others composed of the CPU computation time and the time the CPU waits for GPU, memory and I/O given by the following [138]:

$$t_{frame}[i] = \frac{c_{CPU}}{f_{CPU}} + \frac{c_{Mem}}{f_{Mem}} + \frac{c_{IO}}{f_{IO}} + \frac{c_{GPU}}{f_{GPU}} + \ldots,$$

where c_{CPU}, c_{Mem}, c_{IO} and c_{GPU} are the workload in cycles and f_{CPU}, f_{Mem}, f_{IO} and f_{GPU} are the frequency of CPU, memory, I/O and GPU respectively. Increasing the

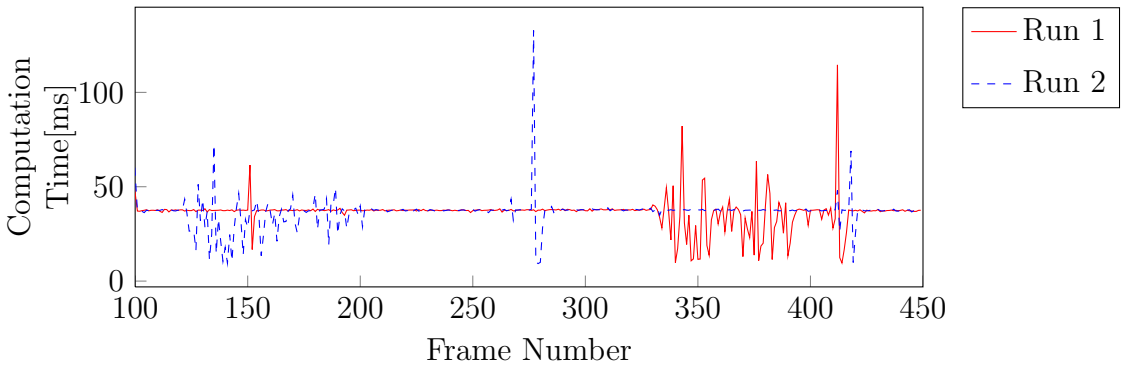


Figure 5.2: Workload of two identical 3D Benchmark runs

processor's frequency will only decrease the time it takes to process the CPU workload, given by the term c_{CPU}/f_{CPU} without affecting the times required for memory accesses, I/O operations and GPU processing. The linear Equation 5.1 only holds true for purely CPU-bound workloads. The relative performance

$$s = \frac{f_{CPU}}{f'_{CPU}} \times \frac{c'[i+1]}{c[i+1]} = \frac{t'[i+1]}{t[i+1]}$$

describes the CPU-boundedness of an application (see Section 2.3.2). As can be seen in Table 5.1, the relative performance of games differs significantly from the relative performance of a purely CPU-bound workload. While the pessimistic purely CPU-bound model from Equation 5.1 would for example assume a slow-down of 3.43 if the frequency is scaled from 1.2 GHz to 350 GHz, in average the processing is only slowed down by a factor of 1.77 in the case of Cut the Rope. Hence, a power manager that has predicted the next frame's workload and has to choose among the available frequencies, can scale the frequency more aggressively if it as well considers the relative performance. Table 5.1 only provides the average factor by which the duration of a frame is being prolonged if a smaller than the highest frequency is used. In reality, s will highly vary from frame to frame, since some frames might be GPU or memory bound, i.e., the CPU has to wait for a non-negligible amount of time for the GPU or memory, whereas other frames might be highly CPU-bound.

Several approaches have been presented in the literature to determine the relative performance factor based on CPU performance counters [24, 25, 138]. These counters capture CPU events, like the number of cache misses, the number of cycles the CPU has to wait for memory, etc.. For benchmarks, taken amongst others from MiBench [55], a relation between performance counter values of the CPU used in this study and the relative performance factor could be found according to [138]. The model, however, neglects GPU- and I/O-bound phases, resulting in substantial errors if applied to gaming applications. To the best of our knowledge, all existing work only leverages benchmarks which utilize the CPU and memory, but does not consider GPU or I/O workload.

To develop such a model, it is required to replay the same workload at different CPU processing frequencies and compare the performance counters. We have developed our

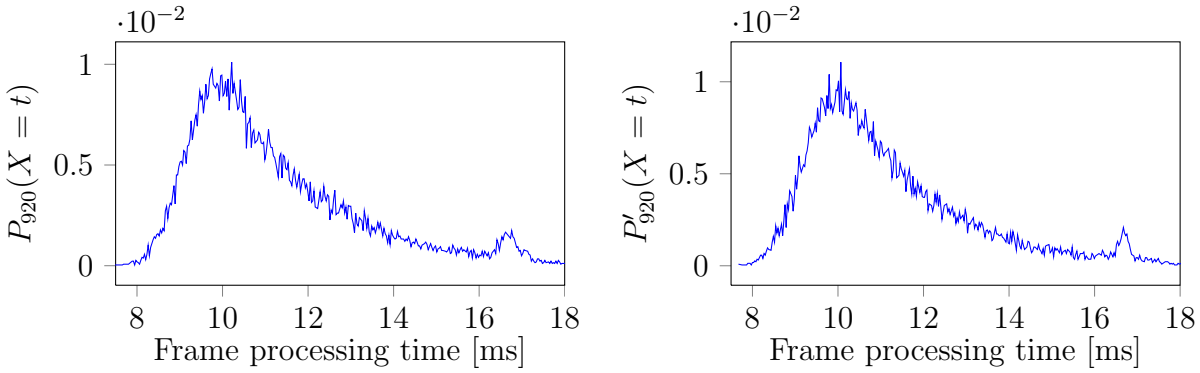


Figure 5.3: Workload histogram of two game plays of Temple Run at 920 MHz each

own gaming benchmark, which animates a knight walking through a 3D landscape, to derive more accurate models that include I/O and GPU waiting times. The input events to this benchmark are generated automatically to guarantee the reproducibility of the workload. Figure 5.2 shows the frame-based workload of two consecutive benchmark runs using a constant CPU processing frequency. Even though, the benchmark has always been started exactly one second after Android has been booted, significant differences can be observed already after 100 frames. Computations of the Android OS and other processes running in Android, which are not necessarily synchronized with the game, heavily disturb the reproducibility. Clearly, variations to this extent do not allow us to accurately tune a model describing the relative performance factor.

Above described difficulties can be avoided using a statistical model. This model does not allow determining the exact optimal sequence of frequencies, but still provides an accurate estimation of the optimal power manager's performance and hence is sufficient for our purposes.

5.1.2 Statistical model

This work targets to determine the minimum possible power consumption that can theoretically be obtained when for each frame the minimum possible frequency is used (considering the target frame rate). In Section 5.1.3, we will show that the power consumption can already be determined, once it is known to what percentage $P(f = f_x)$ each processing frequency $f_x \in \mathcal{F}$ has to be used for processing a game play under consideration of timing constraints. In the following, we describe how it is possible to determine these percentages $P(f = f_x)$ based on game workload recordings.

We have recorded the frame processing times of different game plays using fixed frequencies f_x. From these recordings, we computed probability distributions $P_{f_x}(X = t)$ of the frame processing time. Even though Temple Run randomly generates its game scenarios and hence the content significantly differs between consecutive runs, the two histograms depicted in Figure 5.3 are nearly identical. This could be observed for all available process-

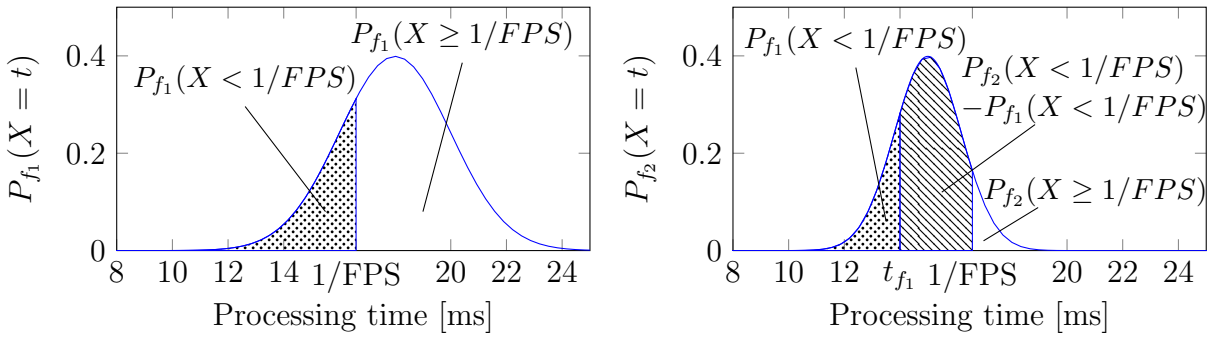


Figure 5.4: Probability distribution of frame-based workloads for two different processing frequencies f_1 and f_2

ing frequencies and games under test (Cut the Rope, Temple Run and Jetpack Joyride) if the games were played long enough (game plays of 10 minutes turned out to be sufficient).

Figure 5.4 shows two examples of probability distributions $P_{f_1}(X = t)$ and $P_{f_2}(X = t)$ recorded at the frequencies f_1 and f_2, respectively. Here, f_1 represents the smallest and f_2 the next higher of all available CPU clock frequencies.

The probability $P(f = f_1)$ that a frame can be finished within the deadline $1/FPS$ if frequency f_1 is used is given by

$$P(f = f_1) = P_{f_1}(X < 1/FPS) = \int_0^{1/FPS} P_{f_1}(X = t)\, dX.$$

Further, it can be said that the percentage of frames that will require a higher frequency than f_1 is given with $P_{f_1}(X \geq 1/FPS)$.

Like for P_{f_1}, the probability that a frame can be complete execution in time using f_2 is given with $P_{f_2}(X < 1/FPS)$. Frames that can be completed within time using f_1 will certainly finish if f_2 is being used since increasing the frequency will never increase the processing time. Hence, the area defined by $P_{f_1}(X < 1/FPS)$ is included in $P_{f_2}(X < 1/FPS)$. Consequently, the percentage of frames that will *exactly* require f_2 is given by

$$P(f = f_2) = P_{f_2}(X < 1/FPS) - P_{f_1}(X < 1/FPS).$$

The remaining percentages $P(f = f_3), \ldots, P(f = f_i)$ can be computed accordingly, once the probability distributions of all processing frequencies have been determined. Note, that in Figure 5.4 the area defined by $P_{f_1}(X < 1/FPS)$ is depicted as leftmost area for visualization purposes. The workload composition and the relative performance factor of the frames determine how this area is distributed within $P_{f_2}(X < 1/FPS)$. In the following we explain, how these probabilities can be used to compute the minimum possible power consumption of the theoretical optimal power manager.

5.1.3 Power consumption model

The average power consumption $\overline{P}$ can be approximated as follows: In addition to the frame processing time $t[i]$, we measure the power consumption $\overline{P}_{f_x}[i]$ of each individual frame. Based on these recordings, the average power consumption of frames at frequency f_x is given by

$$\overline{P}_{f_x} = \frac{1}{N} \sum \overline{P}_{f_x}[i].$$

The overall average power consumption can then be approximated by

$$\overline{P} = \sum_{\forall f_i \in F} \overline{P}_{f_i} \cdot P(f = f_i), \tag{5.2}$$

where $P(f = f_i)$ is the extent to which the frequency f_i has to be used to satisfy the processing deadline $1/FPS$. The accuracy of this approximation will be discussed in Section 5.3.

To compare the quality of game power managers, typically two metrics are used: The average power consumption $\overline{P}$ and the percentage of frames D missing their deadline. Based on determined probability distributions, D can be simply determined, since $D = P_{f_{max}}(X \geq 1/FPS)$, i.e., the percentage of frames that cannot be processed in time, even though the highest processing frequency f_{max} is used.

5.1.4 DVFS overhead

The overhead for dynamic voltage and frequency scaling has to be considered in terms of the time and energy that the voltage frequency transition costs. As shown in Section 4.5.1, the overhead highly depends on the current and the target frequency of the processor. However, using the statistical model presented in Section 5.1.2 it is only possible to estimate the percentage of frames that require a particular frequency and not the exact *order* of the processing frequencies, i.e., the optimal frequency sequence $\mathcal{S}$. Hence, it is not possible to consider the exact scaling overhead. In the worst case, we switch at each frame from the current frequency to the frequency which will cause the largest overhead in terms of switching time. This worst case can be modeled by reducing the deadline of $1/FPS$ by the corresponding scaling time. On the contrary, the best case assumes that only a minimum number of frequency switches occur. As will be shown in Section 5.3, there is only a small difference between the best and worst case. Before above described models are verified and applied, we will first explain the experimental setup.

5.2 Experimental setup

Since the phone's processor is configured such that the performance counter values cannot be read, the experimental results of this chapter are based on a different platform. In the

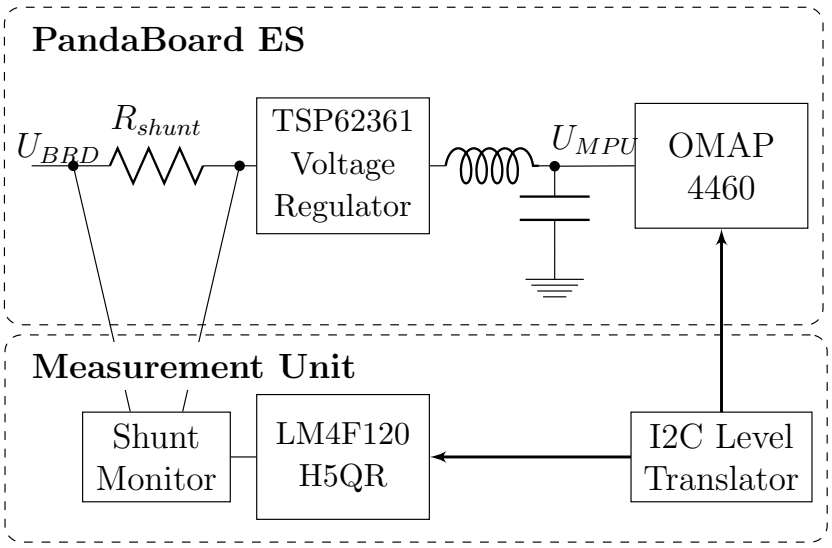


Figure 5.5: Schematic of the measurement setup

following we are first going to describe the details about the hardware platform used for this study before explaining the software modifications.

As a reference board, we used the PandaBoard ES [113] running a Linaro Android 4.3 distribution which uses a 3.2.0 Kernel. Attached to the board is a 10" multitouch LCD display. The PandaBoard hosts the same processor as the Galaxy Nexus, which has been used in Chapter 4, namely the OMAP4460 processor. The CPU subdomain of the OMAP4460 processor is powered by the TPS62361 switching regulator [144]. Due to differences in the PCB layout between the PandaBoard and the Samsung Galaxy Nexus, we could not directly insert the shunt after the TSP62361, but instead placed the shunt before the voltage regulator (see Figure 5.5). Hence, power measurements presented in this section include losses generated by the voltage regulator. Further, the PandaBoard runs a Linaro Android from an external SD card, while the Galaxy Nexus runs an original Android Open Source Project (AOSP) Android from the internal flash storage. Due to these differences, the results presented in this chapter cannot be directly compared to results presented in the previous chapter, even though both platforms are based on the same processor. Same as for the Galaxy Nexus setup, the voltage drop at the shunt and the supply voltage are sampled by a microcontroller (in this case a Texas Instruments Stellaris LM4F120H5QR microcontroller [148]) and can be read by the processor using the I2C interface.

We have instrumented the Linaro Android and Kernel to measure frame timings, to detect different game states and ported the `game state specific` governor to the PandaBoard. Using the state detection, we cannot only provide power management limits for different target frame rates, but as well for individual game states (see Section 5.3). In addition to the power measurement interface, we have extended the Kernel with a performance counter driver, since the used Kernel version did not support this. We have implemented a driver that allows resetting, reading and configuring the performance monitoring unit

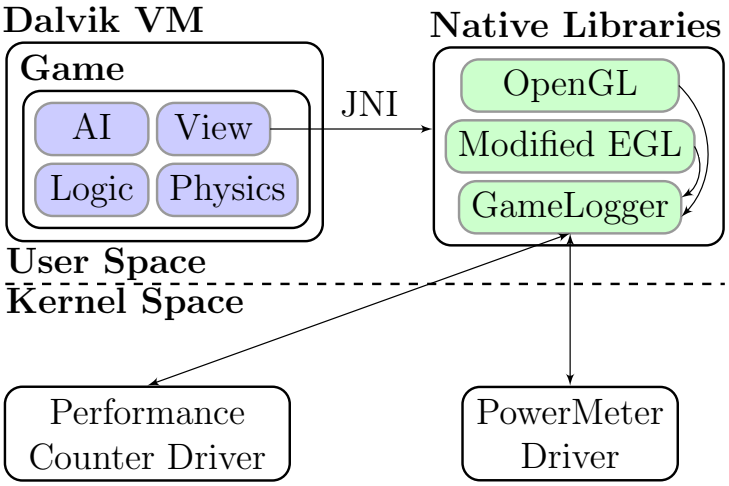


Figure 5.6: Android instrumentation

(PMU) from the user space. To avoid large overheads, the interface has been implemented such that power and performance counter values could be read using a single `ioctl` call (see Figure 5.6). For each frame, the acquired information could be stored in a log file on the SD card.

5.3 Experimental results

Before presenting the results using the statistical model, we first discuss overheads and validate the power model presented in Section 5.1.

5.3.1 Frequency scaling overhead

As described in Section 5.1.4, the overhead for dynamic voltage and frequency scaling has to be considered in terms of the time that is spent on scaling and the energy that the voltage frequency transition costs. The switching overhead of the OMAP4460 has already been presented and discussed in Section 4.5.1. As discussed, based on the statistical model, it is impossible to determine the exact overhead since it does not provide the exact order in which processing frequencies are chosen. However, the resulting power consumption assuming worst and best case (maximum and minimum number of scalings) differed at maximum only by 4.93 % for the game Cut the Rope. The percentage of frames missing the deadline at maximum deviated by 0.69 %. We consider this difference as negligible and hence do not account for ranges in the following, but instead only provide the worst case results. Instrumenting Android, performing the workload prediction, logging data and detecting game states comes with an additional overhead. As has been shown in Section 4.5.1, these overheads can be neglected compared to the workload of a game frame.

Table 5.2: Power consumption of the AR- and oracle-based power manager at different frame rates including the savings that theoretically are possible

Game	State	20 FPS AR [mW]	20 FPS Oracle [mW]	20 FPS Savings [%]	30 FPS AR [mW]	30 FPS Oracle [mW]	30 FPS Savings [%]
Temple Run	Menu	295.0	255.4	13.4	323.3	257.9	20.2
	Gaming	268.3	265.7	1.0	263.3	272.3	-3.4
Jetpack Joyride	Menu	330.1	308.9	6.4	343.1	322.7	5.9
	Gaming	309.1	305.2	1.3	301.7	315.1	-4.4
Cut the Rope	Menu	282.9	287.4	-1.6	317.3	298.8	5.8
	Gaming	256.9	251.7	2.0	273.7	262.4	4.1

Game	State	40 FPS AR [mW]	40 FPS Oracle [mW]	40 FPS Savings [%]	50 FPS AR [mW]	50 FPS Oracle [mW]	50 FPS Savings [%]
Temple Run	Menu	397.8	269.5	32.3	541.1	350.5	35.2
	Gaming	339.9	319.6	6.0	615.8	434.5	29.4
Jetpack Joyride	Menu	566.8	456.7	19.4	672.2	504.8	24.9
	Gaming	349.5	373.6	-6.9	551.7	449.2	18.6
Cut the Rope	Menu	613.8	507.8	17.3	640.6	542.3	15.3
	Gaming	371.0	302.7	18.4	464.4	369.3	20.5

5.3.2 Power model validation

We have performed the following experiment to evaluate the power model described in Section 5.1.3. We first played each game using the `userspace` governor at every available frequency and computed the average power consumption $\overline{P}_{f_x}$ based on these recordings. Next, we played each of the three games using the `game state specific` governor described in Chapter 4 and recorded the frequencies chosen, as well as the real average power consumption of each frame. Based on these recordings, we computed the probability for each frequency to be used during the game play $P(f = f_i)$ and the resulting average power consumption according to Equation 5.2. The maximum deviation of the estimated from the measured power consumption was observed for the game Cut the Rope in the gaming state with 6.01 %. For Jetpack Joyride the maximum deviation was observed in the menu state with 2.96 % and for Temple Run in the gaming state with 1.26 %. We have assumed that these estimation errors are tolerable.

5.3.3 Performance of the optimal power manager

As described in Section 5.1, the optimal power manager chooses for each frame the lowest possible processing frequency that still guarantees the frame computations to complete in

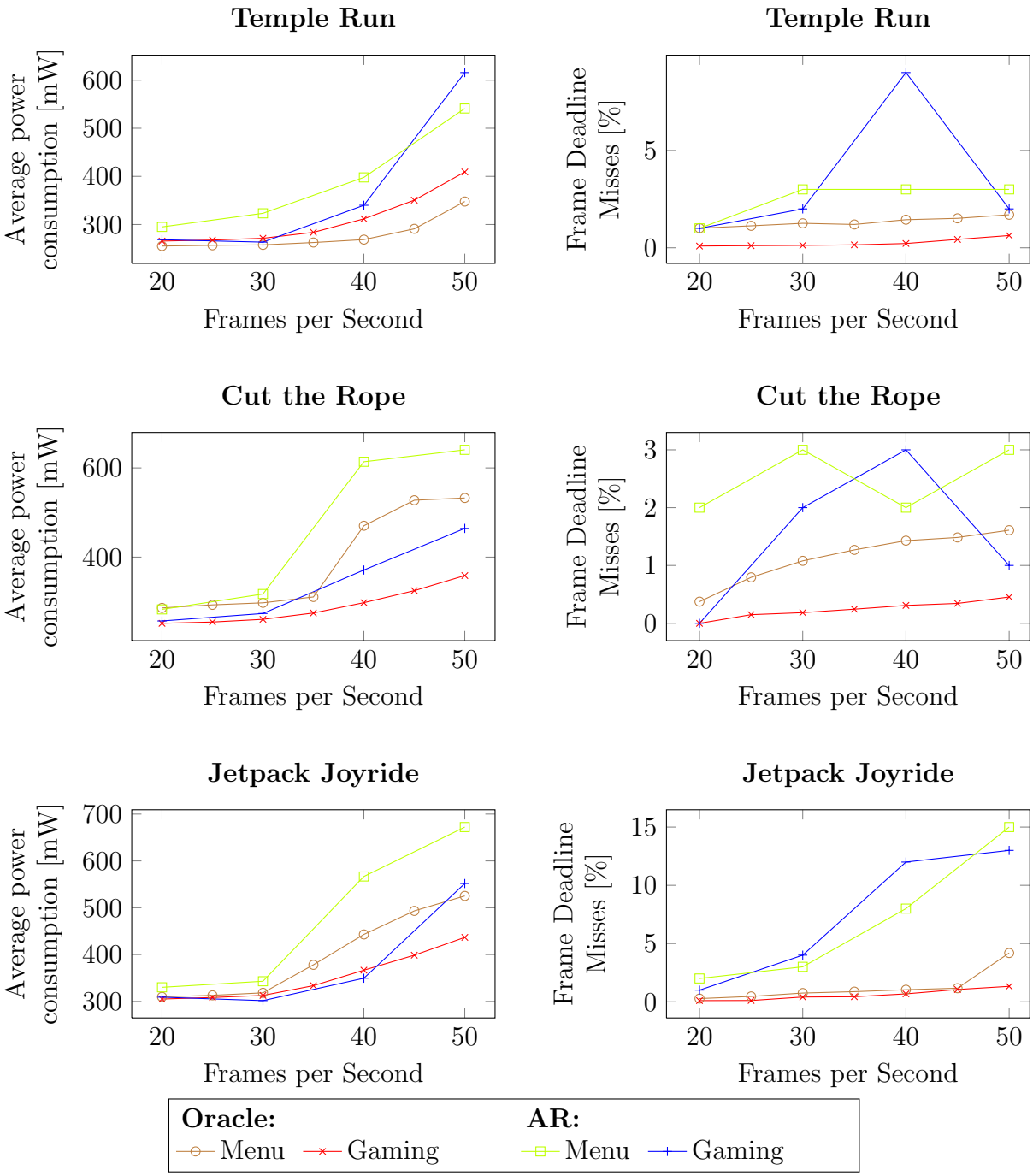


Figure 5.7: Average power consumption and frame deadline misses for different target frame rates using the oracle predictor

time, resulting in a minimum power consumption. The optimal power manager only exists in theory. In practice, the future workload and required processing frequency for a frame have to be predicted, e.g., using AR-based prediction. To determine the gap between practice and theoretical optimum, we played each game using the AR-based predictor and evaluated the statistical model described in Section 5.1.2.

Table 5.2 gives the power consumption for both, the optimal power manager, using the oracle predictor and the power consumption obtained with the AR-based predictor at different frame rates. Further, it provides the theoretically possible power savings expressed as percentage compared to the power consumption obtained with the AR-based power manager. In addition, Figure 5.7 visualizes the power consumption and percentage of frame deadline misses for both approaches. Note that as pointed out in Section 5.2, the power measurement results cannot be compared to measurements presented in Chapter 4, since the results of this chapter are based on the PandaBoard, which, in contrast to the Samsung Galaxy Nexus, allowed reading the performance counter values.

The possible savings in percent were computed according to

$$Savings = 100 \times \frac{P_{AR} - P_{Oracle}}{P_{AR}},$$

where P_{AR} and P_{Oracle} are the average power consumptions obtained with the AR-based predictor and the oracle predictor respectively. For example, at 50 frames per second in Temple Run's Menu state the AR-based predictor consumed 541.1 mW, while the oracle predictor theoretically consumes 350.5 mW resulting in possible savings of 35.2 %.

Clearly, it can be seen that especially for high frame rates, there is a significant performance gap between the AR-based predictor and the theoretical optimum. Up to 35.2 % power could be saved if the predictor exactly knew the future. This considerable potential in terms of power savings points out the need for more research directed to accurate game workload predictors. Note that in some cases the AR-based predictor outperforms the optimal power manager in terms of power consumption, e.g., in the case of Jetpack Joyride (at 40 frames per seond) the AR-based governor saves 6.9 % more power in the gaming state than the optimal power manager. However, in all of these cases the number of frames missing their deadline is higher compared to the optimal power manager. It can as well be seen that in some cases there are frame deadline misses, even if the oracle predictor is used. Here, the maximum possible processing frequency is still not sufficient to guarantee the desired frame rate. As discussed in Chapter 4, for some games the menu is drawn directly on top of the gaming scene, explaining why for these games the menu consumes more power than the actual gaming scene. We assume that the increase of frame deadline misses for the games Temple Run and Cut the Rope in case the target frame rate is reduced from 50 to 40 frames per second is due to the enforced synchronization to the display and a resulting unfortunate timing for these two games, but we were not able to exactly pin-point this issue.

Reducing the target frame rate has a considerable impact on the power consumption for both, the AR-based and the optimal power manager. For example, a reduction from 50 to

Table 5.3: Average power consumption of race-to-halt and the autoregressive model-based power manager

Game	Race-to-halt		AR-based PM	
	Menu [mW]	Gaming [mW]	Menu [mW]	Gaming [mW]
Temple Run	816.7	852.8	332.33	577.71
Jetpack Joyride	1101.9	934.9	343.44	501.43
Cut the Rope	931.36	703.21	342.00	432.53

40 frames per second already reduces the power consumption by 44.9 % for the AR-based predictor in the case of Temple Run, while further reducing the frame rate to 30 frames per second yields power savings of 23.5 % for the AR-based predictor. This finding highly motivates a user study to identify the actually required target frame rates that satisfy the user.

5.3.4 Race-to-halt

Race-to-halt is popularly believed to be an efficient power management strategy and alternative to DVFS. While DVFS tries to minimize the idle time of the CPU, race-to-halt maximizes this time by running the processor at the maximum speed and putting it into a sleep state once a task is processed. In the case of game power management, this approach can be applied on a frame-by-frame-basis: Frames are processed at the highest frequency. Once the processing is done, i.e., when `eglSwapBuffers()` is called, the processor is put into a sleep state until the next VSYNC. The main advantages of this method are that i) no prediction is required and ii) the number of frames missing the deadline is minimized since the frames are processed at the highest frequency. To analyze the resulting power consumption of this approach, we have instrumented Android to measure the energy consumption from the beginning of a frame to the entry point of `eglSwapBuffers()`. Table 5.3 shows the average power consumption of the three games using the race-to-halt approach in comparison to DVFS using AR-based power management. Even though we assume that the processor does not consume any energy in the sleep state, the power consumption of race-to-halt significantly exceeds the power consumption obtained with dynamic voltage and frequency scaling. Hence, we conclude that for the processor model used in this study race-to-halt is not a suitable choice for game power management.

5.4 Summary

DVFS is a widely used approach to reduce a processor's power consumption. Key to an effective power management is an accurate prediction of future workloads. While related work and the work presented in this thesis could show significant savings over existing

methods, an important question was left open: How much power could theoretically be saved, assuming a perfect predictor that exactly *knows* the future workload of a game. In this work, we answer this important question based on a statistical model and not only provide results for games from different genres and frame rates, but as well for different game states. The results of this work have several implications:

Race-to-halt is not a viable alternative to DVFS on current mobile processors due to significantly higher power consumption, even when neglecting the processor's power consumption while idling.

Even though existing work [38, 39] showed considerable power savings compared to Android's default power manager, there is still a substantial gap between AR-based predictors and the theoretical optimum (up to 35.2 % of savings are still possible). This points towards future research endeavors to investigate how current workload prediction can be further improved. More complex statistical models such as non-linear models might be one option to increase the prediction accuracy. Another possible solution might be the classification of frame workloads based on detailed information about the game's internal state obtained from OpenGL and syscall patterns. A standardized interface in Android would, in addition, allow game developers to provide valuable information that could be leveraged by the power manager to further decrease the gap.

The relative performance factor describes the CPU-boundedness of applications. We have shown that for gaming applications the average relative performance factor considerably differs from the worst case, i.e., a purely CPU-bound workload. Hence, current power manager could scale the frequency more aggressively if the relative performance factor was known. Towards this, benchmarks have to be designed that allow identifying the relationship between the relative performance factor and system metrics like the CPU performance counter values or information from the Kernel's process file system entries (like `/proc/pid/io`).

We have shown that already a minor reduction of the frame rate can yield significant power savings for the AR-based predictor. This fact highly motivates a user study, investigating to what extent frame rates can be lowered without deteriorating the by the user perceived quality. Here, aspects like the device's display size, game genre, game state and the player's experience should be taken into consideration.

6

Conclusions and future work

This thesis proposes a power manager that is specific to game applications and leverages dynamic scaling of the mobile processor's voltage and and frequency. Traditional approaches, like the Linux `ondemand` or `interactive` governor, as well use DVFS to reduce the power consumption, but select future frequencies merely based on the past utilization of the processor. These governors are completely unaware of running applications and their requirements, resulting in a highly inefficient power management scheme when games are being played. In this work, we instead propose a power manager that is aware of the game application and considers frame timing requirements as well as the user requirements such as the target frame rate when taking the decision about future processing frequencies. The main contributions and results of this work can be summarized as follows.

- **Workload prediction**

  Key to an efficient power management is the knowledge of future workloads. In Chapter 3, we have formulated requirements for a workload predictor and systematically analyzed different prediction techniques. Using this technique, we reveal that the PID controller-based prediction, which has been successfully applied to video and game applications in the past, is not suitable for game workload prediction since it requires to tune the parameters of the predictor for each individual game play to avoid a diminished prediction performance, resulting in increased power consumption and a deteriorated user experience. Due to the interactive and

non-deterministic nature of games, such tuning on a game play basis is not practical and hence two other approaches are investigated: The least mean squares linear predictor was found to perform well for Quake II workloads, but not for modern games with higher workload variations. We show that the most general form of linear models, namely autoregressive models, provide accurate prediction results, not only when tuned for individual game plays, but as well across game plays and even games. Based on this highly robust prediction scheme we have achieved significant power savings over Linux default governors.

- **Graphics API instrumentation**

  To overcome past restrictions to old closed-source games like Quake II, we have developed a graphics API-based technique that allows applying game power management to modern closed-source games. On Windows-based platforms we leverage DLL injection to intercept the game's communication with the DirectX rendering API. For mobile Android platforms, we presented a direct lightweight instrumentation of the OpenGL ES libraries to obtain the same statistical data, based on which the power manager predicts future workloads.

- **Game state detection**

  Games typically consist of several states like the initial loading, the main menu, level selection menu and the actual gaming states. Each of these states has its characteristic workload and performance requirements. For example, the loading state is likely to be memory-bound, allowing a frequency reduction without prolonging loading times while during menu states, lower target frame rates are likely to be tolerated by the user. We have developed a technique to detect different states. The technique is again only based on the graphics API calls the game issues and hence can be applied to any closed-source game on various platforms.

- **Game state-specific power management**

  Using the game state detection, we have analyzed the power consumption of games during different states. We reveal that players spend a significant amount of time and energy in states like the menu state. We consequently proposed state-specific power management schemes that efficiently reduced the power consumption in the individual states. Compared to Android's default `interactive` governor, we have obtained up to 43.2 % of savings.

- **Limits of game power management**

  In the last part of this work, we analyze the theoretical potential of power savings. We derive a statistical model and reveal the remaining gap between results proposed in this work and the so-called oracle predictor, a predictor that exactly knows the future. The results highly motivate future research endeavors to further decrease the gap between current techniques and the optimal.

6.1 Future work

While proposed techniques significantly outperform default OS power managers, Chapter 5 revealed a substantial gap to the theoretical optimal power manager. To further close the gap we believe that approaches described in the following are promising future directions:

- The instrumentation of the graphics API provides a vast amount of information which indirectly describes the game's current state and is likely to be correlated to future processing requirements. For example, rendering calls related to a particular object might allow predicting AI workload related to that object. Due to the large number of calls per frame and the fact that these calls are very game-specific, it should be investigated if automatized learning techniques from the machine learning or data mining domain can be applied to correlate call patterns to workloads. Thereby, the accuracy of current workload predictors could be further increased.
- All of the presented approaches performed a frame-by-frame workload prediction and frequency scaling. A question that has not been answered, yet, is if game power managers might benefit from an intra-frame prediction. If the governor for example detects particular game events it might be beneficial to reconsider the frequency decision, taken at the beginning of the frame by increasing the frequency to guarantee timing constraints or to lower the frequency due to remaining processing time. In this context, a trade-off between the granularity of frequency scaling and the scaling overhead has to be found to optimize the power savings.
- We have shown significant power savings for the CPU, which is one of the main contributors to the total power consumption. Using our techniques in combination with related approaches for other smartphone components, like the GPU, the display and network interfaces as presented in Chapter 2 of this work, would allow a global optimization of the device's energy consumption when games are being played.
- Finally, we believe that a detailed user study would allow considerable optimization of current techniques. While we have shown that our power management does not impact the perceived gaming quality, it is not clear what exact frame rates and percentage of frames missing their deadline are tolerated by the player. In a user study several aspects and their influence on the perceived quality should be taken into consideration: Display size, game genre, touch latency and for example the user's playing experience. Once the relation between frame rate, frame deadline misses and the quality perceived by the user is known, the power manager can be optimized to a guarantee minimal power consumption while maintaining a good gaming experience.

List of Figures

List of Tables

Bibliography

[1] *Qualcomm Snapdragon Processors Holiday Survey Key Facts.* Qualcomm, 2012.

[2] *Allensbacher Computer- und Technik-Analyse (ACTA) 2013.* Allensbach, 2013.

[3] 148Apps.biz: *App Store Metrics*, (`http://148apps.biz/app-store-metrics`), 2012.

[4] Akenine-Moller, T., E. Haines and N. Hoffman: *Real-Time Rendering.* A K Peters, Ltd., 3rd edition, 2008.

[5] Akyol, E. and M. V. D. Schaar: *Complexity Model Based Proactive Dynamic Voltage Scaling for Video Decoding Systems.* IEEE Transactions on Multimedia, 9(7):1475–1492, 2007.

[6] Anand, B., A. L. Ananda, M. C. Chan, L. T. Le and R. K. Balan: *Game Action Based Power Management for Multiplayer Online Game.* In *Workshop on Networking, Systems, and Applications for Mobile Handhelds (MobiHeld)*, 2009.

[7] Anand, B., A. L. Ananda, M. C. Chan and R. K. Balan: *ARIVU: Making Networked Mobile Games Green - A Scalable Power-Aware Middleware.* Mobile Networks and Applications (MONET), 17(1):21–28, 2012.

[8] Anand, B., K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan and R. K. Balan: *Adaptive Display Power Management for Mobile Games.* In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[9] Android Open Source Project: *MonkeyRunner*, (`http://developer.android.com/tools/help/monkeyrunner_concepts.html`).

[10] Ang, K., G. Chong and Y. Li: *PID Control System Analysis, Design, and Technology.* IEEE Transactions on Control Systems Technology, 13(4):559–576, 2005.

[11] Aydin, H., R. Melhem, D. Mossé and P. Mejía-Alvarez: *Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics.* In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2001.

[12] Aydin, H., R. G. Melhem, D. Mossé and P. Mejía-Alvarez: *Power-Aware Scheduling for Periodic Real-Time Tasks.* IEEE Transactions on Computers, 53(5):584–600, 2004.

[13] Barrio, D., V. Moya, C. González, J. Roca, F. A and E. Roger: *ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures*. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006.

[14] Benini, L., A. Bogliolo and G. De Micheli: *A Survey of Design Techniques for System-Level Dynamic Power Management*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 8(3):299–316, 2000.

[15] Brockwell, P. J. and R. A. Davis: *Introduction to Time Series and Forecasting*. Springer-Verlag, 8th edition, 2010.

[16] Cao, Z. and B. Foo: *Optimality and Improvement of Dynamic Voltage Scaling Algorithms for Multimedia Applications*. IEEE Transactions on Circuits and Systems, 57(3):681–690, 2010.

[17] Carroll, A. and G. Heiser: *An Analysis of Power Consumption in a Smartphone*. In *USENIX Annual Technical Conference*, 2010.

[18] Chang, N., I. Choi and H. Shim: *DLS: Dynamic Backlight Luminance Scaling of Liquid Crystal Display*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 12(8):837–846, 2004.

[19] Chen, K.-c., H.-t. Li and A.-y. A. Wu: *LMS-Based Adaptive Temperature Prediction Scheme for Proactive Thermal-Aware Three-Dimensional Network-on-Chip Systems*. In *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2014.

[20] Chen, X., Y. Chen, Z. Ma and F. C. A. Fernandes: *How is Energy Consumed in Smartphone Display Applications?* In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2013.

[21] Chen, X., J. Zheng, Y. Chen, M. Zhao and C. J. Xue: *Quality-Retaining OLED Dynamic Voltage Scaling for Video Streaming Applications on Mobile Devices*. In *Design Automation Conference (DAC)*, 2012.

[22] Cheng, W., Y. Hou and M. Pedram: *Power Minimization in a Backlit TFT-LCD Display by Concurrent Brightness and Contrast Scaling*. In *Design, Automation and Test in Europe (DATE)*, 2004.

[23] Choi, K., 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)*, 2002.

[24] Choi, K., W. Lee, R. Soma and M. Pedram: *Dynamic Voltage and Frequency Scaling under a Precise Energy Model Considering Variable and Fixed Components of the System Power Dissipation*. In *International Conference on Computer-Aided Design (ICCAD)*, 2004.

[25] Choi, K., R. Soma and M. Pedram: *Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Tradeoff Based on the Ratio of Off-Chip Access to On-Chip Computation Times*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(1):18–28, 2005.

[26] Choi, K., R. Soma and M. Pedram: *Dynamic Voltage and Frequency Scaling Based on Workload Decomposition*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[27] Clark, L. T., N. Deutscher, S. Demmons and F. Ricci: *Standby Power Management for a 0.18µm Microprocessor*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[28] Claypool, M., K. Claypool and F. Dama: *The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games*. In *Multimedia Computing and Networking (MMCN)*, 2006.

[29] Coskun, A. K., T. S. Rosing and K. C. Gross: *Proactive Temperature Balancing for Low Cost Thermal Management in MPSoCs*. In *International Conference on Computer-Aided Design (ICCAD)*, 2008.

[30] Cricenti, A. and P. Branch: *ARMA(1,1) Modeling of Quake4 Server to Client Game Traffic*. In *SIGCOMM Workshop on Network and System Support for Games*, 2007.

[31] Das, S., D. Roberts and S. Lee: *A Self-Tuning DVS Processor Using Delay-Error Detection and Correction*. IEEE Journal of Solid-State Circuits, 41(4):792–804, 2006.

[32] Dhiman, G. and T. Rosing: *Dynamic Voltage Frequency Scaling for Multi-Tasking Systems Using Online Learning*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[33] Dickey, D. A. and W. A. Fuller: *Distribution of the Estimators for Autoregressive Time Series With a Unit Root*. Journal of the American Statistical Association, 74(366):427–431, 1979.

[34] Dietrich, B., S. Nunna, D. Goswami, S. Chakraborty and M. Gries: *LMS-Based Low-Complexity Game Workload Prediction for DVFS*. In *International Conference on Computer Design (ICCD)*, 2010.

[35] Dietrich, B. and S. Chakraborty: *Managing Power for Closed-Source Android OS Games by Lightweight Graphics Instrumentation*. In *Workshop on Network and Systems Support for Games (NetGames)*, 2012.

[36] Dietrich, B. and S. Chakraborty: *DEMO: Power Management Using Game State Detection on Android Smartphones*. In *International conference on Mobile systems (MobiSys)*, 2013.

[37] Dietrich, B. and S. Chakraborty: *Forget the Battery, Let's Play Games!* In *Symposium on Embedded Systems For Real-Time Multimedia (Estimedia)*, 2014.

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

[39] Dietrich, B., D. Goswami, S. Chakraborty, A. Guha and M. Gries: *Time Series Characterization of Gaming Workload for Runtime Power Management.* IEEE Transactions on Computers (due to appear), 2014.

[40] Diniz, B. and D. Guedes: *Limiting the Power Consumption of Main Memory.* In *International Symposium on Computer Architecture (ISCA)*, 2007.

[41] Donald, J. and M. Martonosi: *Techniques for Multicore Thermal Management: Classification and New Exploration.* SIGARCH Computer Architecture News, 34(2):78–88, 2006.

[42] Dong, M., Y.-S. K. Choi and L. Zhong: *Power-Saving Color Transformation of Mobile Graphical User Interfaces on OLED-Based Displays.* In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[43] Dropsho, S., V. Kursun, D. Albonesi, S. Dwarkadas and E. Friedman: *Managing Static Leakage Energy in Microprocessor Functional Units.* In *International Symposium on Microarchitecture (MICRO)*, 2002.

[44] Duan, R., M. Bi and C. Gniady: *Exploring Memory Energy Optimizations in Smartphones.* In *International Green Computing Conference and Workshops (IGCC)*, 2011.

[45] Ericsson: *Mobility Report - On the Pulse of the Networded Society.* 2014.

[46] Euh, J. and W. Burleson: *Exploiting Content Variation and Perception in Power-Aware 3D Graphics Rendering.* In *Power-Aware Computer Systems*, pages 51–64. Springer, 2001.

[47] Farago, P.: *The Truth About Cats and Dogs: Smartphone vs Tablet Usage Differences*, (`http://blog.flurry.com/bid/90987`), 2012.

[48] Gleixner, T. and D. Niehaus: *Hrtimers and Beyond: Transforming the Linux Time Subsystems.* In *Proceedings of the Linux Symposium*, 2006.

[49] Govil, K., E. Chan and H. Wasserman: *Comparing Algorithm for Dynamic Speed-Setting of a Low-Power CPU.* In *International Conference on Mobile Computing and Networking (MobiCom)*, 1995.

[50] Gu, Y.: *Power Management for Interactive 3D Games.* PhD thesis, National University of Singapore, NUS, Singapore, 2008.

[51] Gu, Y. and S. Chakraborty: *A Hybrid DVS Scheme for Interactive 3D Games.* In *Real-Time Technology and Applications Symposium (RTAS)*, 2008.

[52] Gu, Y. and S. Chakraborty: *Control Theory-Based DVS for Interactive 3D Games*. In *Design Automation Conference (DAC)*, 2008.

[53] Gu, Y. and S. Chakraborty: *Power Management of Interactive 3D Games Using Frame Structures*. In *International Conference on VLSI Design (VLSID)*, 2008.

[54] Gu, Y., S. Chakraborty and W. T. Ooi: *Games Are Up for DVFS*. In *Design Automation Conference (DAC)*, 2006.

[55] Guthaus, M. and J. Ringenberg: *MiBench: A Free, Commercially Representative Embedded Benchmark Suite*. In *International Workshop on Workload Characterization (WWC)*, 2001.

[56] Ha, Y., J. Yi, H. Horii and J. Park: *An Edge Contact Type Cell for Phase Change RAM Featuring Very Low Power Consumption*. In *Symposium on VLSI Technology*, 2003.

[57] Haase, C. and R. Guy: *For Butter or Worse - Smoothing Out Performance in Android UIs*, (`http://youtu.be/Q8m9sHdyXnE`), Google IO 2012.

[58] Haykin, S.: *Adaptive Filter Theory*. Prentice Hall, 1991.

[59] Heweltt-Packard, Intel, Microsoft, Phoenix Technologies and Toshiba: *Advanced Configuration and Power Interface Specification*. 2013.

[60] Hicks, P., M. Walnock and R. Owens: *Analysis of Power Consumption in Memory Hierarchies*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 1997.

[61] Hoglund, G. and G. McGraw: *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley Professional, 2007.

[62] Hong, I., G. Qu, M. Potkonjak and M. B. Srivastava: *Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors*. In *Real-Time Systems Symposium (RTSS)*, 1998.

[63] Hosseini, M., A. Fedorova, J. Peters and S. Shirmohammadi: *Energy-Aware Adaptations in Mobile 3D Graphics*. In *International Conference on Multimedia*, 2012.

[64] Hu, Z., A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson and P. Bose: *Microarchitectural Techniques for Power Gating of Execution Units*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[65] Hua, S. and G. Qu: *Approaching the Maximum Energy Saving on Embedded Systems with Multiple Voltages*. In *International Conference on Computer-Aided Design (ICCAD)*, 2003.

[66] Huang, Y., S. Chakraborty and Y. Wang: *Watermarking Video Clips with Workload Information for DVS*. In *International Conference on VLSI Design (VLSID)*, 2008.

[67] Hughes, C. J., K. Praful, S. V. Adve, J. Rohit, C. Park and S. Jayanth: *Variability in the Execution of Multimedia Applications and Implications for Architecture*. In *International Symposium on Computer Architecture (ISCA)*, 2001.

[68] Hughes, C. J., J. Srinivasan and S. V. Adve: *Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications*. In *International Symposium on Microarchitecture (MICRO)*, 2001.

[69] Imagination Technologies: *POWERVR MBX Technology Overview*. 2009.

[70] Iranli, a., W. Lee and M. Pedram: *HVS-Aware Dynamic Backlight Scaling in TFT-LCDs*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14(10):1103–1116, 2006.

[71] Iranli, A., W. Lee and M. Pedram: *Backlight Dimming in Power-Aware Mobile Displays*. In *Design Automation Conference (DAC)*, 2006.

[72] Isci, C., A. Buyuktosunoglu and M. Martonosi: *Long-Term Workload Phases: Duration Predictions and Applications to DVFS*. IEEE Micro, 25(5):39–51, 2005.

[73] Isci, C., G. Contreras and M. Martonosi: *Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management*. In *International Symposium on Microarchitecture (MICRO)*, 2006.

[74] Ishihara, T. and H. Yasuura: *Voltage Scheduling Problem for Dynamically Variable Voltage Processors*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 1998.

[75] Jejurikar, R. and R. Gupta: *Dynamic Voltage Scaling for Systemwide Energy Minimization in Real-Time Embedded Systems*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[76] Kao, J.: *Subthreshold Leakage Control Techniques for Low Power Digital Circuits*. PhD thesis, Massachusetts Institute of Technology, 2001.

[77] Keating, M., D. Flynn, R. Aitken, A. Gibbons and K. Shi: *Low Power Methodology Manual for System-on-Chip Design*. 2007.

[78] Khronos Group: *EGL*, (`www.khronos.org/egl`).

[79] Khronos Group: *OpenGL ES*, (`www.khronos.org/opengles`).

[80] Kim, D., N. Jung and H. Cha: *Content-Centric Display Energy Management for Mobile Devices*. In *Design Automation Conference (DAC)*, 2014.

[81] Kim, H., N. Agrawal and C. Ungureanu: *Revisiting Storage for Smartphones*. ACM Transactions on Storage, 8(4):1–25, 2012.

[82] Kwiatkowski, D., P. C. Phillips, P. Schmidt and Y. Shin: *Testing the Null Hypothesis of Stationarity Against the Alternative of a Unit Root: How Sure Are We that Economic Time Series Have a Unit Root?* Journal of Econometrics, 54(1-3):159–178, 1992.

[83] Laurens, P., R. F. Paige, P. J. Brooke and H. Chivers: *A Novel Approach to the Detection of Cheating in Multiplayer Online Games.* In *International Conference on Engineering Complex Computer Systems*, 2007.

[84] Lee, C., M. Potkonjak and W. H. Mangione-Smith: *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems MediaBench Components.* In *International Symposium on Microarchitecture (MICRO)*, 1997.

[85] Lee, W., K. Patel and M. Pedram: *White-LED Backlight Control for Motion-Blur Reduction and Power Minimization in Large LCD TVs.* Journal of the Society for Information Display, 17(1):37, 2009.

[86] Liang, W.-Y., S.-C. Chen, Y.-L. Chang and J.-P. Fang: *Memory-Aware Dynamic Voltage and Frequency Prediction for Portable Devices.* In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[87] Liang, W.-y. and P.-t. Lai: *Design and Implementation of a Critical Speed-Based DVFS Mechanism for the Android Operating System.* In *International Conference on Embedded and Multimedia Computing (EMC)*, 2010.

[88] Lin, B., A. Mallik, P. A. Dinda, G. Memik and R. P. Dick: *User- and Process-Driven Dynamic Voltage and Frequency Scaling.* In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[89] Lin, C.-h., C. Kang and P. Hsiu: *Catch Your Attention: Quality-Retaining Power Saving on Mobile OLED Displays.* In *Design Automation Conference (DAC)*, 2014.

[90] Lin, W. and C.-C. Jay Kuo: *Perceptual Visual Quality Metrics: A Survey.* Journal of Visual Communication and Image Representation, 22(4):297–312, 2011.

[91] Liu, X., P. Shenoy and W. Gong: *A Time Series-Based Approach for Power Management in Mobile Processors and Disks.* In *International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2004.

[92] Ljung, L.: *System Identification: Theory for the User.* Prentice-Hal PTR, 1999.

[93] Lorch, J. and A. Smith: *PACE: A New Approach to Dynamic Voltage Scaling.* IEEE Transactions on Computers, 53(7):856–869, 2004.

[94] Lu, Z., J. Lach and M. Stan: *Reducing Multimedia Decode Power Using Feedback Control.* In *International Conference on Computer Design (ICCD)*, 2003.

[95] Ma, X., Z. Deng, M. Dong and L. Zhong: *Characterizing the Performance and Power Consumption of 3D Mobile Games*. IEEE Computer, 46(4):76–82, 2013.

[96] Malik, A., B. Moyer and D. Cermak: *A Low Power Unified Cache Architecture Providing Power and Performance Flexibility*. In *International symposium on Low power electronics and design (ISLPED)*, 2000.

[97] Mallik, A., B. Lin, G. Memik, P. A. Dinda and R. P. Dick: *User-Driven Frequency Scaling*. IEEE Computer Architecture Letters, 5(2):16–19, 2006.

[98] Marchesan, A. G., R. Busseuil, E. Alceu Carara, N. Hebert, S. Varyani, G. Sassatelli, P. Benoit, L. Torres and F. Gehm Moraes: *Predictive Dynamic Frequency Scaling for Multi-Processor Systems-on-Chip*. In *International Symposium on Circuits and Systems (ISCAS)*, 2011.

[99] Marple, S. L.: *A New Autoregressive Spectrum Analysis Algorithm*. IEEE Transactions on Acoustics, Speech, and Signal Processing, 28(4):441–454, 1980.

[100] Marple Jr., S. L.: *Digital Spectral Analysis with Applications*. Prentice Hall, 1987.

[101] McShaffry, M.: *Game Coding Complete*. 2009.

[102] Meindl, J. and J. Davis: *The Fundamental Limit on Binary Switching Energy for Terascale Integration (TSI)*. IEEE Journal of Solid-State Circuits, 35(10):1515–1516, 2000.

[103] Microsoft: *Microsoft Detours*, (`http://research.microsoft.com/en-us/projects/detours`), 2010.

[104] Mochocki, B., K. Lahiri and S. Cadambi: *Power Analysis of Mobile 3D Graphics*. In *Design, Automation and Test in Europe (DATE)*, 2006.

[105] Mochocki, B., K. Lahiri, S. Cadambi and X. S. Hu: *Signature-Based Workload Estimation for Mobile 3D Graphics*. In *Design Automation Conference (DAC)*, 2006.

[106] Montgomery, D., C. Jennings and M. Kulahci: *Introduction to Time Series Analysis and Forecasting*. John Wiley & Sons, Inc., 2008.

[107] Newzoo Games Market Research: *2011 Games Market Report*. 2011.

[108] Newzoo Games Market Research: *2012 Country Summary Report US*. 2012.

[109] NVIDIA: *G-SYNC*, (`http://www.geforce.com/hardware/technology/g-sync`).

[110] Pallipadi, V. and A. Starikovskiy: *The Ondemand Governor - Past, Present, Future*. In *Linux Symposium*, 2006.

[111] Pallipadi, V., S. Li and A. Belay: *cpuidle: Do Nothing, Efficiently*. In *Proceedings of the Linux Symposium*, 2007.

[112] Panda, P. R., B. V. N. Silpa, A. Shrivastava and K. Gummidipudi: *Power-Efficient System Design*. Springer US, 2010.

[113] Pandaboard.org: *Pandaboard ES - System Reference Manual*. 2011.

[114] Park, J., D. Shin, N. Chang and M. Pedram: *Accurate Modeling and Calculation of Delay and Energy Overheads of Dynamic Voltage Scaling in Modern High-Performance Microprocessors*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2010.

[115] Pathania, A., Q. Jiao, A. Prakash and T. Mitra: *Integrated CPU-GPU Power Management for 3D Mobile Games*. In *Design Automation Conference (DAC)*, 2014.

[116] Pedram, M.: *Power Minimization in IC Design: Principles and Applications*. ACM Transactions on Design Automation of Electronic Systems, 1(1):3–56, 1996.

[117] Pering, T. and R. Brodersen: *Energy Efficient Voltage Scheduling for Real-Time Operating Systems*. In *Real-Time Technology and Applications Symposium (RTAS)*, 1998.

[118] Pering, T., T. Burd and R. Brodersen: *The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms*. In *International Symposium on Low power electronics and design (ISLPED)*, 1998.

[119] Pillai, P. and K. Shin: *Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems*. ACM SIGOPS Operating Systems Review, 35(5):89–102, 2001.

[120] Pool, J.: *Energy-Precision Tradeoffs in the Graphics Pipeline*. PhD thesis, University of North Carolina, 2012.

[121] Pouwelse, J., K. Langendoen and H. Sips: *Dynamic Voltage Scaling on a Low-Power Microprocessor*. In *Internactional Conference on Mobile Computing and Networking (MobiCom)*, 2001.

[122] Powell, M., B. Falsafi, K. Roy and T. Vijaykumar: *Gated-V_{dd}: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories*. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.

[123] Quan, G. and X. Hu: *Minimal Energy Fixed-Priority Scheduling for Variable Voltage Processors*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22(8):1062–1071, 2003.

[124] Q.Wu, P. Juang, M. Martonosi, L.-S. Peh and D. W. Clark: *Formal Control Techniques for Power-Performance Management*. IEEE Micro, 25(5):52–62, 2005.

[125] Ranganathan, P., E. Geelhoed, M. Manahan and K. Nicholas: *Energy-Aware User Interfaces and Energy-Adaptive Displays*. IEEE Computer, 39(6):31–38, 2006.

[126] Rele, S., S. Pande, S. Onder and R. Gupta: *Optimizing Static Power Dissipation by Functional Units in Superscalar Processors.* In *Compiler Construction*, pages 261–275. Springer, Berlin Heidelberg, 2002.

[127] Said, S. E. and D. A. Dickey: *Testing for Unit Roots in Autoregressive-Moving Average Models of Unknown Order.* Biometrika, 71(3):599–607, 1984.

[128] Sedgewick, R.: *Algorithms in C - Parts 1-4: Fundamentals, Data Structures, Sorting, Searching.* Addison-Wesley-Longman, Third edition, 1998.

[129] Shim, H., N. Chang and M. Pedram: *A Backlight Power Management Framework for Battery-Operated Multimedia Systems.* IEEE Design & Test of Computers, 21(5):388–396, 2004.

[130] Shin, D., Y. Kim, N. Chang and M. Pedram: *Dynamic Voltage Scaling of OLED Displays.* In *Design Automation Conference (DAC)*, 2011.

[131] Siddha, S., V. Pallipadi and A. Ven: *Getting Maximum Mileage Out of Tickless.* In *Linux Symposium*, 2007.

[132] Silpa, B. V. N., G. Krishnaiah and P. R. Panda: *Rank Based Dynamic Voltage and Frequency Scaling for Tiled Graphics Processors.* In *International Conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, 2010.

[133] Silpa, B. V. N., A. Patney, T. Krishna, P. R. Panda and G. S. Visweswaran: *Texture Filter Memory - A Power-Efficient and Scalable Texture Memory Architecture for Mobile Graphics Processors.* In *International Conference on Computer-Aided Design (ICCAD)*, 2008.

[134] Sinha, A. and A. Chandrakasan: *Dynamic voltage scheduling using adaptive filtering of workload traces.* In *International Conference on VLSI Design (VLSID)*, 2001.

[135] Sinha, A. and A. Chandrakasan: *Dynamic Power Management in Wireless Sensor Networks.* IEEE Design & Test of Computers, 18(2):62–74, 2001.

[136] Skadron, K., T. Abdelzaher and M. Stan: *Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management.* In *International Symposium on High Performance Computer Architecture (HPCA)*, 2002.

[137] Snowdon, D. C., S. Ruocco and G. Heiser: *Power Management and Dynamic Voltage Scaling: Myths and Facts.* In *Workshop on Power Aware Real-time Computing*, 2005.

[138] Snowdon, D., S. Petters and G. Heiser: *Accurate On-line Prediction of Processor and Memory Energy Usage under Voltage Scaling.* In *Proceedings of the 7th ACM IEEE International Conference on Embedded Software (EMSOFT)*, 2007.

[139] Snowdon, D., E. L. Sueur, S. M. Petters and G. Heiser: *Koala: A Platform for OS-Level Power Management.* In *European Conference on Computer Systems (EuroSys)*, 2009.

[140] Srinivasan, V. and G. Shenoy: *Energy-Aware Task and Interrupt Management in Linux.* In *Linux Symposium*, 2008.

[141] Sueur, E. L. and G. Heiser: *Slow Down or Sleep, That Is the Question.* In *USENIX Annual Technical Conference*, 2011.

[142] Tan, K., T. Okoshi, A. Misra and R. Balan: *FOCUS: A Usable & Effective Approach to OLED Display Power Management.* In *International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2013.

[143] Téllez, G., A. Farrahi and M. Sarrafzadeh: *Activity-Driven Clock Design for Low Power Circuits.* In *International Conference on Computer-Aided Design (ICCAD)*, 1995.

[144] Texas Instruments: *3A Processor Supply with I2C Compatible Interface and Remote Sense, TPS62362 Datasheet.*

[145] Texas Instruments: *Mixed Signal Controller, MSP430F551x Datasheet.*

[146] Texas Instruments: *Voltage Output, High or Low Side Measurement, Bi-Directional Zero-Drift Current Shunt Monitor, INA199A2 Datasheet.*

[147] Texas Instruments: *OMAP4460 Multimedia Device, OMAP4460 Datasheet*, 2012.

[148] Texas Instruments: *Stellaris LM4F120H5QR Microcontroller*, 2013.

[149] Thirugnanam, K., B. Anand, J. Sebastian, P. G. Kannan, A. L. Ananda, R. K. Balan and M. C. Chan: *Dynamic Lookahead Mechanism for Conserving Power in Multi-Player Mobile Games.* In *INFOCOM*, 2012.

[150] Varma, A., B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan and J. Bruce: *A Control-Theoretic Approach to Dynamic Voltage Scheduling.* In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2003.

[151] Veendrick, H.: *Short-Circuit Dissipation of Static CMOS Circuitry and its Impact on the Design of Buffer Circuits.* IEEE Journal of Solid-State Circuits, 19(4):468–473, 1984.

[152] VESA: *www.vesa.org.*

[153] Weiser, M., B. Welch, A. Demers and S. Shenker: *Scheduling for Reduced CPU Energy.* In *USENIX Conference on Operating Systems Design and Implementation*, 1994.

[154] Wu, Q., P. Juang, M. Martonosi and D. W. Clark: *Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors.* In

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2004.

[155] Wu, Q., M. Pedram and X. Wu: *Clock-Gating and Its Application to Low Power Design of Sequential Circuits*. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 47(3):415–420, 2000.

[156] Yao, F., A. Demers and S. Shenker: *A Scheduling Model for Reduced CPU Energy*. In *Foundations of Computer Science (FOCS)*, 1995.

[157] Yuan, L. and G. Qu: *Analysis of Energy Reduction on Dynamic Voltage Scaling-Enabled Systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(12):1827–1837, 2005.

[158] Yuan, W. and K. Nahrstedt: *Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems*. In *Symposium on Operating systems principles (SOSP)*, 2003.

[159] Yun, H.-S. and J. Kim: *On Energy-Optimal Voltage Scheduling for Fixed-Priority Hard Real-Time Systems*. ACM Transactions on Embedded Computing Systems, 2(3):393–430, 2003.

[160] Zhai, B., D. Blaauw, D. Sylvester and K. Flautner: *Theoretical and Practical Limits of Dynamic Voltage Scaling*. In *Design Automation Conference (DAC)*, 2004.

[161] Ziegler, J. G. and N. B. N. Rochester: *Optimum Settings for Automatic Controllers*. Amercian Society of Mechanical Engineers, 64(11):759–765, 1942.