Games are Up for DVFS

Yan Gu Samarjit Chakraborty Wei Tsang Ooi
Department of Computer Science
National University of Singapore
{guyan,samarjit,ooiwt}@comp.nus.edu.sg

ABSTRACT

Graphics-intensive computer games are no longer restricted to high-performance desktops, but are also available on a variety of portable devices ranging from notebooks to PDAs and mobile phones. Battery life has been a major concern in the design of both the hardware and the software for such devices. Towards this, dynamic voltage and frequency scaling (DVFS) has emerged as a powerful technique. However, the showcase application for DVFS algorithms so far has largely been video decoding, primarily because it is computationally expensive and its workload exhibits a high degree of variability. This paper investigates the possibility of applying DVFS to interactive computer games, which to the best of our knowledge has not been studied before. We show that the variability in the workload associated with a popular First Person Shooter game like Quake II is significantly higher than video decoding. Although this variability makes game applications an attractive candidate for DVFS, it is unclear if DVFS algorithms can be applied to games due to their interactive (and hence highly unpredictable) nature. In this paper, we show using detailed experiments that (surprisingly) interactive computer games are highly amenable to DVFS. Towards this we present a novel workload characterization of computer games, based on the game engine for Quake II. We believe that our findings might potentially lead to a number of innovative DVFS algorithms targeted towards game applications, exactly as video decoding has motivated a variety of schemes for DVFS.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; C.4 [**Performance of systems**]: Design studies

General Terms

Experimentation, Measurement, Performance

Keywords

Dynamic Voltage and Frequency Scaling, Computer Games, Computer Graphics, Animation, Multimedia, Graphics Workload Characterization, Power-aware Design

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

1. INTRODUCTION

Computer games have recently experienced a sharp increase in popularity and have attracted considerable attention in both the industry and the academia. They are driving a number of innovations in areas ranging from graphics hardware and high-performance computer architecture to networking and software engineering. Although most of the graphics-rich games are still largely played on high-performance desktops, over the last couple of years a number of games are also available on portable devices such as PDAs (e.g. `www.doompda.com`). Since such devices are becoming increasingly popular and powerful, this trend will certainly continue.

Energy efficiency is one of the most critical issues in the design of such battery-powered portable devices. The availability of dynamic voltage and frequency-scalable processors has lead to power management schemes for portable devices that are based on dynamic voltage and frequency scaling (DVFS) algorithms. Since the power consumed by a processor depends linearly on its frequency and on the square of its operating voltage, DVFS algorithms scale the operating frequency and voltage of the processor to match a varying computational workload as closely as possible. The showcase application for DVFS algorithms so far has largely been video decoding [2, 4, 5, 6], primarily because of two reasons: (i) video decoding applications are computationally expensive, and (ii) their workload exhibits high variability. These reasons make video decoding applications ideal candidates to illustrate the potential energy savings that may be achieved by DVFS algorithms. A number of innovative DVFS schemes have indeed been motivated by video decoding applications.

Contributions of this paper: Motivated by the abovementioned line of work and the increasing availability of game applications on portable devices, this paper attempts to answer the following natural questions. Are interactive game applications amenable to DVFS? Is the workload associated with game applications sufficiently variable, so that DVFS schemes are worthwhile? Can we predict the workload of game applications so that we can scale the operating frequency and voltage of the processor to match the workload? To answer these questions, we have carried out detailed experiments using an open source, popular First Person Shooter game called Quake II [8]. Our experiments studied the changes in processor cycle requirements for different frames as the game is played, and the distribution of these processor cycles among the different computation and rendering tasks.

We performed our measurements based on Quake II for several reasons. First, it was the only open source, main stream, First Person Shooter game available when we started our project[1]. Second, Quake II belongs to an older generation of games, which can be

[1]Source code for Quake III has since been released.

played without a powerful CPU and special hardware accelerators. Since we are targeting portable devices such as PDAs, where special graphics hardware is not likely to be available, we believe it is a representative game that can be played on current, general purpose portable devices. Finally, the *game engine* of Quake II is the basis of other popular First Person Shooter games. Here, we would like to clarify that a *game engine* is the reusable core of a game application. By adding details (which are often referred to as "assets") like models, animation, sound and story to a game engine, a (concrete) game is derived. Since our experimental results are based on Quake II, they immediately extend to other First Person Shooter games (e.g., Hexen II) derived from the same game engine. We believe that our results serve as a good starting point for research in this area and they would also be valid for many other game applications that are based on different game engines, but have similar software architectures.

Given the interactive and hence unpredictable nature of game applications, apparently it might seem that they are not amenable to DVFS. However, our results show that surprisingly games are highly amenable to DVFS. Further, the nature of their workload is very different from those arising from video decoding applications, which motivate the need for different DVFS schemes compared to the ones traditionally used for video decoding. There are two fundamental differences between the workloads arising from game and video decoding applications:

- The *magnitude* of the variation in the number of processor cycles required to *process* a frame is significantly higher in the case of games compared to video decoding. However, the *frequency* of this variation is much higher in the case of video decoding applications. In other words, compared to the workload arising from a game, a video decoding workload exhibits a smaller but more rapid variation. This observation indicates that the potential energy savings that may be obtained from applying DVFS to games is higher than what may be obtained from video decoding applications.

- In the case of game applications, the frames contain "structure" which can be exploited to predict their workload or processor cycle requirements. While processing a frame, the workload depends heavily on the scene that the frame is depicting. More specifically, the workload depends on the content of the frame or the constituting *objects* that need to be processed. We discuss this issue further in detail in Section 4. In contrast, video frames offer much less structure, apart from their frame type (I, B, or P frames).

The rest of this paper mostly elaborates on the above two arguments and presents experimental results to support them. Following these arguments, our main contribution is a novel workload characterization of graphics-intensive game applications. We believe that based on this workload characterization, it will be possible to develop several DVFS schemes that are targeted towards computer games.

Organization of this paper: In the next section we discuss the architecture of common game engines. In Section 3 we investigate the possibility of lowering the frame-rate of a game, thereby reducing its processing workload. This reduction would enable the game application to run at a constant, but lower processor frequency, thereby reducing power consumption. Although this approach would be a competing approach to DVFS, we discuss what are its disadvantages and why dynamically changing the processor's voltage/frequency might be better in the case of games. In Section 4 we present a framework for characterizing the workload of games and use it to discuss how DVFS schemes might be designed for games. We also outline how such schemes might differ from those used in the context of video decoding. Finally, in Section 5 we list a number of possible directions in which this work may be extended.

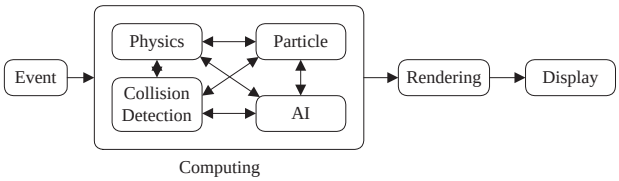


Figure 1: Frame *processing* in a game application.

2. ANATOMY OF A GAME ENGINE

A game engine runs in an infinite loop, where the body of this loop consists of tasks responsible for processing a single frame. This loop body is shown in Figure 1. Here *Event* denotes the user inputs or interactions with the game, which along with the current state of the game is used to generate the next frame to be displayed. This involves two sequential steps—computing and rendering—which we describe below. A more detailed discussion may be found in [1, 10].

The computing step comprises tasks such as collision detection, AI, simulation of game physics and particle systems. Collision detection includes algorithms for checking collisions between the different objects and characters in the game. Such algorithms compute intersections between two given solids, their trajectories as they move, impact times during a collision and their impact points. In some engines, the AI tasks determine the movement of the characters in the game. Game physics incorporates physical laws into the game engine so that different effects (e.g. collisions) appear more realistic to a player. Typically, simulation physics is only a close approximation of real physics, and computation is performed using discrete rather than continuous values. Finally, a particle system model allows a variety of other physical phenomenon to be simulated. These include smoke, moving water, blood, explosions and gun fires. The number of particles that may be simulated are typically restricted by the computing power of the machine on which the game is being played.

The rendering step involves algorithms to generate an image (or a frame) from a model, which is then displayed as shown in Figure 1. In this case, the model is typically a description of several three dimensional objects using a predefined language or data structure. It consists of geometry, viewpoint, texture and lighting information. In the case of 3D graphics, rendering may be done offline, as in pre-rendering, or in real time. Pre-rendering is a computationally intensive process that is typically used for movie creation, while real-time rendering is commonly done in 3D computer games, which often rely on the use of a specialized processor called a Graphics Processing Unit (GPU).

The rendering steps include the transformation of the vertices of solid objects to the screen space, deletion of invisible pixels by clipping, rasterization, deletion of occluded pixels, and interpolation of various parameters. The outcome of these steps is the transformation of the 3D data onto the 2D screen. Rendering is computationally expensive and occupies a significant fraction of the total processing time of a frame.

3. A FIRST CUT: REDUCING FRAME RATES

A rule of thumb in game design is that users prefer high frame rates. As a result, most game applications are designed to maximize frame rates without any consideration towards resource usage or power consumption. The loop described in Section 2 therefore runs at the maximum possible rate and fully utilizes the available CPU bandwidth. We measured the CPU usage of Quake II running on a 1298 MHz notebook computer using the Intel VTune Analyzer 7.2 [9] and noticed that it occupies 95% of the CPU bandwidth on an average. However, the frame rate varies over time and depends on the state of the game (e.g. the number of characters and the complexity of the scene).

A recent study [3] on the effects of frame rates and resolution in First Person Shooter games concluded that although frame rates have a significant impact on the perceived quality-of-service, for most parts of a game very high frame rates are not required. More specifically, the resulting frame rate when a game application fully utilizes the CPU bandwidth might be unnecessarily high. As a result, a natural question that comes up is: Why not run the game at a *constant* (but lower) frequency?

It turns out that this is not a good strategy, because the variation in the number of processor cycles required to process different frames is considerably high, as we show in Section 4. While running the CPU at a constant but lower frequency would reduce the overall frame rate, the rate might drop below the tolerable range when rendering complex scenes. Before we present the results supporting this observation, let us briefly outline the experimental setup that we use throughout this paper.

Experimental setup: We conducted all our experiments on an IBM notebook with a 1298 MHz Intel Mobile Processor built with Speedstep technology, 768 MB of DRAM, and an ATI Radeon Mobility Video card with a 144 MHz M6 GPU. The CPU supports five different frequency operating points: 1298, 1199, 999, 799 and 599 MHz. All our results are based on the "vanilla" Quake II, version 3.21, whose source code was instrumented and compiled to run on Windows XP. To ensure that the game is not preempted by other processes, we ran it with the highest priority and rendered the game with the "software" option. The "software" option disables the use of the GPU, causing the 3D functions to be executed on the CPU. This option uses DirectDraw to draw the pixels on the screen. Sounds were disabled during measurements, as our initial results show that the workload in loading and playing audio during games is negligible (approximately 1.8% of the total workload). All the processor cycle measurements were carried out using RDTSC (read time-stamp counter) instruction. We chose to use a software-only renderer as many battery-powered personal mobile devices such as (low-end) laptops, PDAs and mobile phones do not support GPUs yet.

To ensure reproducibility, instead of actually playing the game, we replayed pre-recorded demo files in Quake II. All measurements were done by replaying the default demo file that comes with Quake II unless when indicated otherwise. The game resolution was set to 1024×768, running in full-screen mode. While replaying demos allows us and the research community to repeat our experiments, the workload measured is slightly lower than the workload incurred by games played in real-time. The difference arises from the fact that, the demo has certain pre-recorded states (such as position of objects in each frame and input from users) and therefore these states are not computed again during playback. Our experiments suggest that this computation accounts for approximately 3% of the total workload of the game.

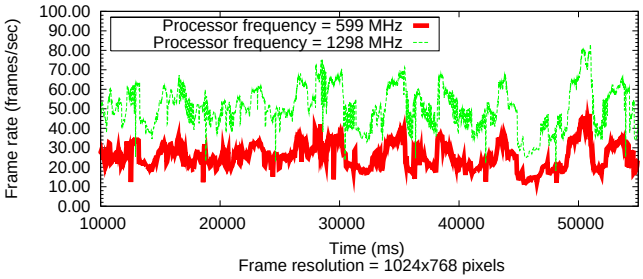


Figure 2: Resulting frame rates when the processor frequency is set to 599 MHz and 1298 MHz.

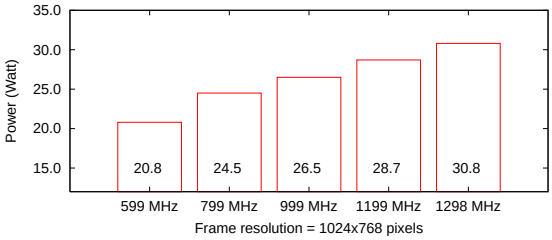


Figure 3: Average power consumption for different processor frequencies.

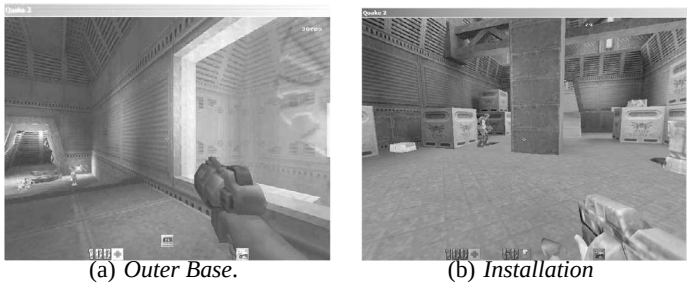
For power measurements, we removed the battery from the notebook and connected it to the external power supply using an AC power adapter. We then tapped the cable leading from the power adapter to the notebook using special probes connected to a digital oscilloscope (DL1540CL Yokogawa) which measured the instantaneous current and voltage drawn by the notebook.

Figure 2 shows an excerpt of how the instantaneous frame rate varies with time for replaying the default Quake II demo with the processor frequency set to 599 MHz and 1298 MHz. We measured the instantaneous frame rate as the reciprocal of the frame processing time. Note that with 1298 MHz, the frame rate varies between approximately 30 and 70 frames per second (fps). With 599 MHz, the frame rate varies roughly between 10 and 40 fps. With frequencies set to values between 599 and 1298 MHz, the frame rates lie between the two plots shown in Figure 2. A frame rate of 70 fps is much higher than necessary [3]. On the other hand, if we run the processor at a constant frequency of 599 MHz, we achieve undesirably low frame rates on certain frames exhibiting complex scenes.

The average power consumptions corresponding to the five frequency values supported by our notebook with the game running on it are shown in Figure 3. We computed these values by recording the instantaneous current $c(t)$ and voltage $v(t)$ drawn by the notebook every 5 ms, and calculating the power consumption over a duration of length T as $\sum_{t=0}^{T}(c(t)v(t)\delta t)/T$, where δt is the sampling interval (5 ms). Note that these values correspond to the total system power consumption and not the power consumed by the processor alone.

4. THE CASE FOR DVFS

Figure 3 shows the potential energy savings that can be achieved by running the processor at a lower (but constant) frequency. However, Figure 2 clearly indicates that by using DVFS, the fluctuations in the frame rate can be reduced, thereby resulting in an acceptable perceptual quality and at the same time a reduced energy consumption.



(a) *Outer Base*. (b) *Installation*

Figure 4: Two game maps in Quake II.

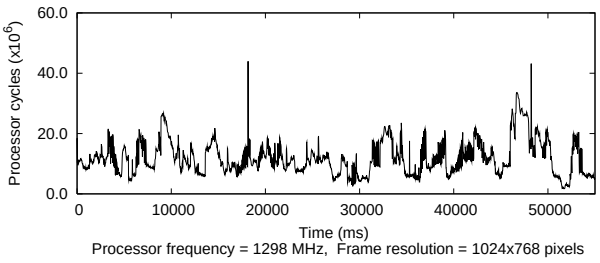


Figure 5: Rasterization workload per frame.

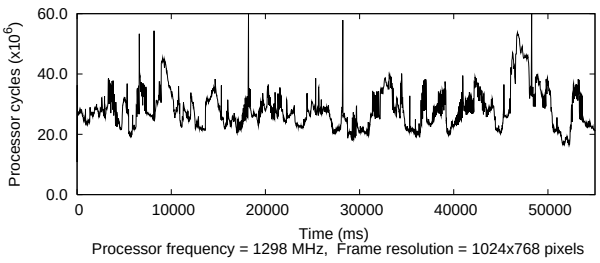


Figure 6: Total processing workload for per frame.

In this section we construct the case for such a DVFS scheme. Towards this, we start by presenting a framework for characterizing the workload of game applications. We then discuss how such a framework might be used to design DVFS algorithms for games. Finally, we outline how such DVFS algorithms would differ from those used for video decoding applications.

4.1 Workload Characterization of Games

As mentioned in Section 2, a game engine is designed to sequentially execute the computing and rendering tasks. For each frame, the engine polls the user's input and passes it over to the computing subsystems responsible for collision detection, AI, particle simulation etc. These subsystems compute new locations and appearances of the visible objects based on the user input. We refer to the resulting workload as the *computation workload*. The results of these computations are passed to the rendering task, which renders all the visible objects in the current frame and displays them on the screen. A significant component of this rendering task involves *rasterizing* objects on the screen. From this point on, we will primarily be concerned with this rasterization component of the rendering task, for reasons which we explain later in this section. Henceforth, we call the workload resulting from the rasterization task as the *rasterization workload.* When Quake II uses its software renderer, all tasks—including geometry processing, rasterization and texture processing—are performed on the CPU.

Before proceeding further, we will need to understand what a *game map* (also referred to as a *level*) is. The storyline of a game can be considered to progress from one location (or level) to the next, where each of these locations is represented using a game map. Examples of game maps might be cities, buildings, rooms and corridors. Intuitively, a game map may be considered to be a data structure which stores all the objects and characters in the scenario represented by the map. Snapshots of two different game maps from Quake II, called *Outer Base* and *Installation* are shown in Figures 4(a) and 4(b) respectively. The game map *Installation* is used in the default demo.

A commonly used data structure to represent a game map is a Binary Space Partition (BSP) tree [10]. A BSP tree represents a recursive, hierarchical partitioning or subdivision of space into convex subspaces. The BSP tree is constructed by partitioning a space using a hyperplane, with the resulting partitions being further partitioned by recursively applying the same procedure. For each leaf in the BSP tree, a set of leaves that are *visible* from this leaf are calculated and updated as the game is played. This set is referred to as the Potentially Visible Set (PVS). In addition, the BSP tree also records information related to texture and lighting. Both the computation and the rendering steps shown in Figure 1 involve traversing and manipulating the BSP tree.

In Quake II, a game map is divided into convex regions, forming the leaves of the BSP tree. To render a game map, first the BSP tree is traversed to determine the leaf in which the *camera* is located Once this leaf is identified, the PVS associated with this leaf lists the potentially visible leaves from this camera location [7]. The bounding box of these leaves are then used to quickly *cull* leaves from the PVS that are not within the viewing frustum. The remaining leaves are then passed to the subsequent rendering tasks, which includes matrix transformations on the data and the rasterizing of a frame as 2D image onto the screen.

4.1.1 Workload as a Function of Scene Complexity

The rasterization workload of a frame clearly has a direct correspondence with the *objects* that are contained in the frame. In other words, it depends on the "complexity" of the scene to be rendered. Figure 5 shows how this rasterization workload changes with time. The total workload involved in processing a frame also has a correspondence with the complexity of the frame. This correspondence can be seen from Figure 6, which shows how the total workload changes with time. Note that the fluctuations in the processor cycle demands in Figures 5 and 6 are highly correlated. Further, our measurements show that the rasterization workload constitutes approximately 38% of the total workload generated in processing a frame.

From these two observations, we believe that one can predict the total processing workload to reasonable accuracy if one can estimate the rasterization workload. The rest of this paper shows how the rasterization workload can be predicted. We propose a workload characterization in which the workload associated with rasterizing a frame depends on the *objects* constituting the frame. Our experimental results show that for Quake II, the type of objects that contribute to this workload are the *brush model*, the *Alias model*, the *texture*, *light maps* and *particles*. Below we discuss the workload characterization of each object type.

Brush Model: A brush model is a 3D convex solid composed of polygons. Brush models are used to construct the geometry of a game map and they define the "world space" in which the player can move around. The workload resulting from rasterizing a frame will depend on the *number* of brush models in the frame and also

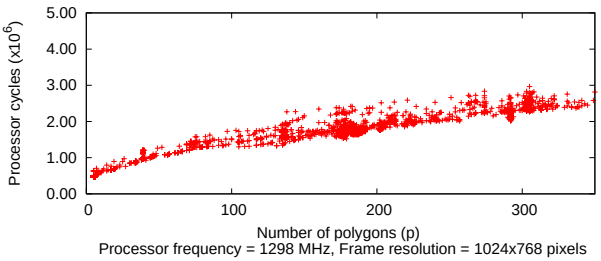


Figure 7: Workload for a brush model versus the number of polygons constituting the brush model (Game Map: *Command Center*).

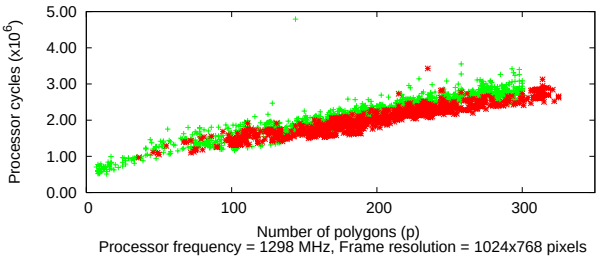


Figure 8: Workload for a brush model versus the number of polygons constituting the brush model (Game Maps: *Outer Base* and *Installation*).

the *types* of these models. We therefore parameterize a brush model using the number of *polygons* constituting it. To identify the workload involved in rasterizing a brush model with a specified number of polygons, we collected the number of polygons constituting each brush model and the number of processor cycles involved in rasterizing them. Our results are shown in Figure 7, which were obtained from the replay of a demo using the game map *Command Center*. The figure shows that the number of processor cycles required to rasterize a brush model increases almost linearly with the number of polygons in it.

Let n_p denote the number of brush models in a frame with p polygons. Let w be the average workload resulting from a single polygon. Then the total workload arising from all the brush models in this frame is equal to $\sum_p n_p \times p \times w$. Parameter w is the slope of the line in Figure 7.

To see if the relationship between the rasterization workload and the number of polygons in a brush model hold for *different* game maps, we repeated the experiments using other game maps as well. Figure 8 shows the workload involved in rasterizing brush models with different number of polygons from two other game maps, *Outer Base* and *Installation* (see Figures 4(a) and 4(b)). Note that the value of w is consistent across the three game maps and the workload involved in rasterizing a brush model also increases linearly with the number of polygons for the latter two game maps.

Alias Model: Alias models are used to represent the different entities in Quake II (such as monsters, soldiers and weapons). Usually an Alias model consists of the geometry and the skin texture of the entity being modeled. The geometry in turn is composed of *triangles*. Since the rasterization of the triangles is done on the CPU instead of a graphics hardware, the number of *pixels* constituting each triangle affects the CPU workload. The software renderer renders the skin texture of an Alias model with two rendering *modes* called *opaque* and *alpha blend*. These modes call different functions and therefore incur different rasterization workloads.

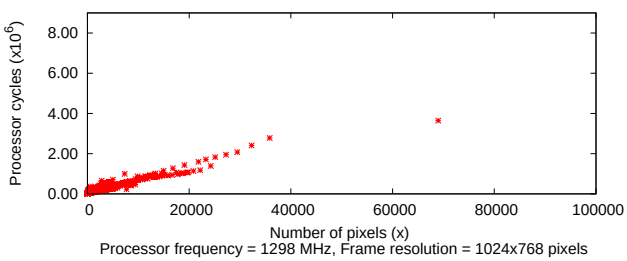


Figure 9: Workload involved in rasterizing Alias models for different values of x (t = 0).

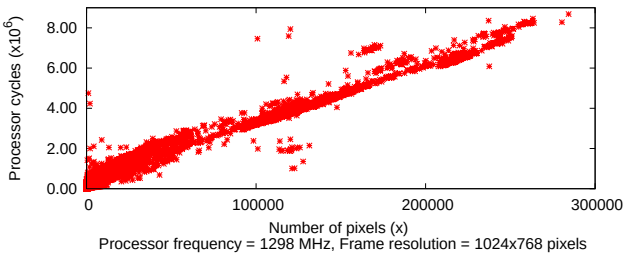


Figure 10: Workload involved in rasterizing Alias models for different values of x (t = 1).

For each mode, we characterize the rasterization workload of an Alias model by the total number of pixels rendered. This number can be obtained by summing up the area of the triangles constituting an Alias model. Let x denote the total number of pixels of an Alias model. Let $t = 0$ denote the case where Alias models with alpha blended texture are being used, and $t = 1$ denote the case where models with opaque texture are being used. To compute the rasterization workload of Alias models with different values of x and t, we capture all the models arising in different frames along with their rasterization workloads. Figure 9 and 10 show the results for Alias models with different x, for $t = 0$ and $t = 1$ respectively.

These figures show that the rasterization workload of Alias models scale almost linearly with x. The same linear relationship also holds for Alias models with different values of t. Further, these relationships are also consistent across game maps.

Texture: Texture is the 2D image applied to the face of a brush model to give it the appearance of a real surface, examples of which are concrete slabs, brick walls and metal plates. A texture is typically composed of multiple *surfaces*. We therefore characterize the rasterization workload of a texture in terms of the number of surfaces constituting it. As in the case of brush models, we capture the textures arising from a sample game play and plot their rasterization workload versus their number of surfaces. In this case we found that the rasterization workload increases almost linearly with the number of surfaces in a texture. Again, this function remains consistent across different game maps.

Light Map: Light maps are used to store pre-calculated lighting information for different scenes in a game. Static light maps in Quake II are low resolution bitmaps which are rendered as multiple *surfaces*. Hence, the workload involved in rasterizing light maps is already included in the workload resulting from rasterizing textures. Therefore it need not be accounted for separately.

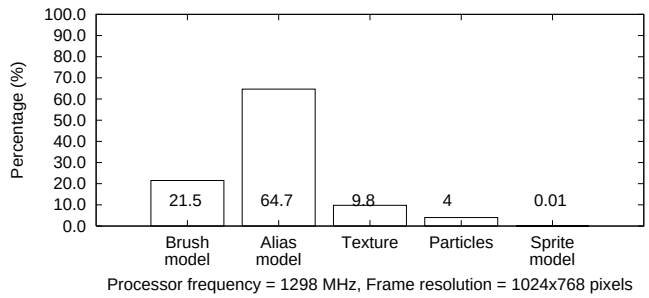


Figure 11: Contributions of the different objects in a frame towards the rasterization workload.

Particles: Particles are often used to model small debris resulting from gun shots hitting a target. They are usually generated as a set of 3D *points*. The number of *pixels* of the points generated in rasterization is used to parameterize the rasterization workload of particles. This workload scales almost linearly with the number of pixels, as expected, and the scaling factor again remains consistent across game maps.

The contributions of the abovementioned five types of objects to the rasterization workload are summarized in Figure 11 (the workload resulting from light maps is not shows for reasons already described). Rasterizing Alias models is clearly the most expensive. Lastly, note that apart from these five objects, *sprite models* are also responsible for a small fraction (almost negligible) of the rasterization workload. These models are often used to represent dust particles or special effects like sparkles.

4.2 Predicting Frame Workload for DVFS

Most DVFS algorithms targeted towards video decoding applications rely on predicting the processing workload of future frames or macroblocks and then adjusting the processor's operating voltage and frequency to match this workload as closely as possible. Such predictions are often based on the decoding times (or equivalently, processor cycle requirements) of previously decoded frames.

As mentioned in Section 1, we believe that DVFS schemes for game applications would require fundamentally different approaches. More specifically, the workload prediction for a frame should *not* rely on the processing times of previous frames. Instead, the "structure" in the frame should be exploited to predict its workload. The framework for workload characterization that we presented in the previous subsection can be used towards this. Using this framework, the rasterization workload of a frame can be computed as the sum of the rasterization workloads of its constituent objects. The computed workload can then be appropriately scaled to *predict* the total processing workload of the frame, which can be used to adjust the processor's voltage and frequency.

While computing, or rather predicting, the rasterization workload of the different objects constituting a frame, several data structures or tables need to be created, as discussed in the previous subsection. An example of such a table is the workload of each (single) Alias model for different values of the parameters (x, t). Exactly how these tables are created, and more importantly how they are maintained or updated would depend on the specifics of the DVFS scheme. In contrast to such schemes, the only "structure" information that DVFS algorithms for video decoding applications can use is whether the frame is of type I, B or P.

Figure 6 shows how the number of processor cycles required to completely process a sequence of frames vary in Quake II. The corresponding variations in the case of video decoding have much smaller magnitude. As discussed in Section 1, this observation points to the potentially greater energy savings that can be achieved in the case of game applications. Finally, we would like to conclude by pointing out that buffering techniques to smooth out the variations in the decoding times of frames, which are widely used in video decoding applications, cannot be used for games due to their interactive nature.

5. CONCLUDING REMARKS

This paper was concerned with building a case for DVFS algorithms specifically targeted towards interactive games. Our main contribution was a framework for characterizing the workload of game applications. Towards this we presented several experimental results using the Quake II game engine. We also outlined how the proposed workload characterization framework may be used to design concrete DVFS algorithms and how such algorithms might differ from those used for video decoding applications.

The logical future work that stems from this paper is to use the proposed framework to *predict* the processor cycle requirements of frames and use this prediction to scale the processor's voltage and frequency. Our initial experimental results show significant system-wide energy savings. However, more work needs to be done to *efficiently* maintain the various tables and compute/predict the frame workloads.

Acknowledgements: Thanks are due to Ying Chee Woo and Chandra Mukaya from the ECE Department of NUS for their help in setting up our experiments for power measurement.

6. REFERENCES

[1] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, 1998.

[2] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 732–737, San Jose, California, 2002.

[3] M. Claypool, 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) Conference*, San Jose, California, 2006.

[4] C. J. Hughes and S. V. Adve. A formal approach to frequent energy adaptations for multimedia applications. In *International Symposium on Computer Architecture (ISCA)*, pages 138–149, Munich, Germany, 2004.

[5] C. Im, S. Ha, and H. Kim. Dynamic voltage scheduling with buffers in low-power multimedia applications. *ACM Transactions in Embedded Computing Systems*, 3(4):686–705, 2004.

[6] Z. Lu, J. Lach, M. R. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *International Conference on Computer Design (ICCD)*, pages 489–497, San Jose, California, 2003.

[7] M. McGuire. Quake 2 BSP File Format http://www.flipcode.com/articles/article_q2bsp.shtml.

[8] Quake II, http://www.idsoftware.com/games/quake/quake2/.

[9] Intel VTune Performance Analyzer http://www.intel.com/cd/software/products/asmona/eng/vtune/vpa/index.htm.

[10] A. Watt and F. Policarpo. *3D Games: Real-time Rendering and Software Technology, Volume 1*. Addison-Wesley, 2001.