Energy Adaptation for Multimedia Information Kiosks

Richard Urunuela
Obasco Group
EMN-INRIA, LINA
Nantes, France
rurunuel@emn.fr

Gilles Muller
Obasco Group
EMN-INRIA, LINA
Nantes, France
gmuller@emn.fr

Julia L. Lawall
DIKU
University of Copenhagen
Copenhagen, Denmark
julia@diku.dk

ABSTRACT

Video kiosks increasingly contain powerful PC-like embedded processors, allowing them to display video at a high level of quality. Such video display, however, entails significant energy consumption. This paper presents an approach to reducing energy consumption by adapting the CPU clock frequency. In contrast to previous approaches, we exploit the specific behavior of a video kiosk. Because a kiosk plays the same set of movies over and over, we choose a CPU frequency for a given frame based on the computational requirements of the frame that were observed on earlier iterations. We have implemented our approach in the legacy video player MPlayer. On a PC like those that can be found in kiosks, we observe increases in battery lifetime of up to 2 times as compared to running at the maximum CPU frequency on a set of high resolution divx movies.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*

General Terms

Algorithms, Performance, Measurement

Keywords

Dynamic voltage scaling, multimedia application, embedded systems

1. INTRODUCTION

Video kiosks are becoming commonplace in bus stops, airports, and other public places where people need entertainment and information. Because such kiosks run continuously, power management is critical, both to reduce costs and to allow the use of limited energy sources such as solar power for outdoor kiosks and battery power for mobile ones. Video kiosks, however, have high computational requirements, as they must be able to display complex videos at a level of quality that meets audience expectations. These computational requirements entail a high rate of CPU power consumption.

An effective strategy for reducing CPU power consumption is to reduce the CPU voltage, which leads to a quadratic energy savings [4]. Nevertheless, reducing the voltage implies a corresponding reduction in the CPU frequency, which slows application execution. Because applications can tolerate such a slowdown to a varying degree, recent processors allow *dynamic voltage scaling* (DVS), *i.e.*, changing the voltage during program execution [10]. A number of power management strategies have been developed around this facility [5, 11, 13, 14, 18, 19]. These strategies dynamically choose the minimum frequency that will allow an application to meet its timing requirements, and have been shown to result in significant energy savings for a variety of applications, including MPEG video [5, 11, 13].

The difficult part of a DVS strategy is to anticipate the computational requirements of the application. General-purpose approaches observe recent CPU load and assume that upcoming load will be similar [5, 17, 18]. Video codecs, however, use compression strategies that imply that frames vary significantly in their complexity, meaning that recent computational requirements are a poor predictor of upcoming behavior [8]. While some success has been achieved for MPEG and MPEG-2 video using such predictive approaches [5], the problem is compounded for modern codecs such as divx, which offer a very high compression ratio. To avoid the problems of generic prediction, other approaches have resorted to modifying the video decoding algorithm or the operating system (OS), in order to obtain more precise information about computational requirements [15, 19]. These approaches, however, are not easily portable, and require substantial expertise to implement.

This paper. In this paper, we propose a new power management approach, History-based DVS (HbDVS), that takes advantage of the specific properties of video kiosks. As compared to general-purpose video players, a video kiosk typically plays the same set of videos over and over. Thus, our approach predicts a frame's computational requirements based not on recent behavior in the current iteration, but on the behavior for the same frame in a previous iteration. We show furthermore that when the video player is the only application, as is the case in a kiosk, the variance in the computation time for a given frame between different iterations of a video is very low. Thus, our approach chooses a frequency for each frame in the early iterations of the video, and then uses the chosen frequency subsequently.

Our approach offers the following advantages:

- It requires adding only a few lines of code to the multimedia player and no modifications to the OS. Thus, it can be integrated easily into legacy systems.
- It is independent of the video format, and thus remains applicable as new, more efficient, formats are developed. In our tests, we use divx videos, as this format is widely used and provides a high compression ratio, reducing the duration of I/O when reading from the disk, which provides further energy savings.
- It is online and thus does not require prior access to the installed hardware. This property is crucial when a video kiosk network uses diverse hardware platforms.
- It is effective on both high resolution and low resolution videos, increasing battery lifetime by up to 109% as compared to the maximal frequency and 40% as compared to the Linux tool `powernowd`, with no perceptible loss of quality.

The rest of the paper is as follows. Section 2 describes previous work on DVS, focusing on approaches that have been applied to multimedia applications. Section 3 investigates properties of divx video that have an impact on power management. Section 4 presents our solution, Section 5 evaluates the resulting energy savings, and Section 6 concludes.

2. RELATED WORK

The two main categories of DVS algorithms are interval-based algorithms and task-based algorithms [16]. In addition, some video-specific approaches have been proposed.

2.1 Interval-based algorithms

Interval-based algorithms monitor the CPU load at various time intervals. According to the observed load, the algorithm changes the CPU frequency and voltage.

One such algorithm is PAST [17], which is implemented in the Linux `powernowd` tool. PAST is based on the assumption that upcoming CPU requirements will be similar to recent ones. Thus, if the previous interval was mostly idle (load under 50%), PAST decreases the CPU speed, and if the previous interval was mostly busy (load over 70%), PAST increases the CPU speed. Variants have been proposed that weight the observed loads in various ways [7].

Interval-based algorithms are typically simple and application independent. Nevertheless, experiments that test these algorithms in practice [8, 13] show that CPU utilization by itself does not provide enough information about application timing requirements to ensure both meeting application quality of service requirements and saving energy.

2.2 Task-based algorithms

While interval-based algorithms consider the entire CPU workload within an interval, task-based algorithms distinguish between the computational requirements of individual tasks.

Vertigo is a task-based voltage manager for Linux [5]. It uses information collected at the OS level to classify tasks as interactive or periodic. For each category of task, Vertigo provides a specific strategy for accumulating a task's recent computational requirements and choosing an appropriate frequency. This approach has been successfully applied to playback of MPEG videos. Nevertheless, by the published measurements, playback of these videos exhibits a large percentage of idle and sleep time, suggesting that they are not as demanding as the divx videos we consider. Furthermore, the approach requires modifications to the OS.

Weissel and Bellosa [18] use hardware events as the basis for choosing the clock frequencies for different process. The motivation is that the rate at which a process generates various hardware events indicates its performance and energy dissipation. Unfortunately, in the case of video, the work done for each frame varies considerably, and thus cannot easily be predicted from the resource requirements of previous frames.

PACE [11] is a strategy for improving existing DVS algorithms by replacing the use of a constant frequency by a speed schedule, which begins with a lower frequency and gradually increases the frequency, if needed. This approach saves energy when a task completes earlier than expected. PACE is well-adapted to applications where computational requirements vary, as is the case for video, but introduces many frequency changes in the more demanding parts of the computation.

GRACE-OS also uses a speed schedule, but uses a video-specific analysis to compute it [19]. GRACE-OS is furthermore built into the process scheduler, and thus requires modifying the OS.

2.3 Video-specific algorithms

Finally, several video-specific approaches have been proposed. A key issue in applying DVS to video is the large variation between the computational requirements of the different frames [8]. These approaches estimate these requirements online, on a frame-by-frame basis.

Burchard and Altenbernd [3] propose to separate the processing of a video into two phases. The first phase decodes enough of each frame to determine the elements it contains, and the second phase completes the decoding by processing each of these elements. Worst-case execution time (WCET) analysis is integrated into the first phase, to estimate the cost of completing the treatment of the various identified elements of each frame in the second phase. This approach requires a major reorganization of the video player.

Pouwelse proposes a process scheduler that performs DVS based on the estimated execution times of each process [15]. For a H.263 video player, he obtains the estimated execution time from a combination of the frame type and the frame size. As H.263 video frames do not contain size information, this must either be calculated by a preprocessing phase or estimated by the player from the decoding of a portion of the frame. Both approaches require knowledge of the video format and the latter also requires modifying the decoder.

Im and Ha [9] observe that latency is not a critical issue in video playback, as long as the frame rate is respected. They thus propose to buffer a few upcoming frames in the player, and to begin the treatment of these buffered frames during the any slack time of the current frame. Because the treatment of a buffered frame can then stretch over a longer period, *e.g.* the slack time of the current frame plus the frame time of the upcoming frame, it can be carried out at

Video	Resolution	Frames/sec.	Frame time (ms.)	Frames	Playing time
Madagascar preview	1280×720	24	41.67	2758	1mn 54 sec
Jarhead preview	1024×728	23.98	41.71	3159	2mn 11sec
Harry Potter and the Goblet of Fire preview	640×272	24	41.67	3379	2mn 20 sec
X-Men 3 preview	420×748	30	33.37	3112	1mn 43 sec

Figure 1: Videos and their properties

a lower frequency. Choosing the frequency and the buffer size, however, requires knowing the WCET of each frame.

Maxiaguine, Chakraborty and Thiele [12] also consider a buffering video player, and adjust the frequency in response to buffer fill levels. The choice of frequency depends on offline WCET analysis complemented with on-line monitoring. This approach again relies on WCET analysis and on a specific strategy for implementing the video player.

3. THE POTENTIAL FOR REDUCTION IN CPU POWER CONSUMPTION

Decoding video using recent divx codec is more computationally intensive than decoding the MPEG video used in previous DVS experiments. Nevertheless, as processor power has increased, we show in this section that there is still substantial room to reduce energy consumption in this case. Our measurements were done on an Intel Pentium 4M based Dell Inspiron laptop, with available frequencies 1700, 1400, 1200, 800 and 600 MHz. Processors such as the Pentium 4M are increasingly being used in embedded systems, and are often necessary to display high-resolution videos. We use the video player MPlayer (`http://www.MPlayerhq.hu`), running under the Linux 2.6.12 kernel.

Figure 1 summarizes the divx videos used in our tests. All were obtained from `http://www.divx.com`. Regardless of the resolution used by the video, all videos are displayed at a resolution of 1400x1050, corresponding to the maximum resolution of the screen of our test machine. These videos contains both static scenes, such as titles, and highly dynamic live-action scenes. This variety evaluates multiple kinds of situations, since the computational requirements of a frame depend on both the resolution and the number of pixels that have changed since the previous frame.

3.1 The effect of frequency adaptation on perceived quality

A power management strategy must allow the application to maintain an appropriate quality of service. To measure the perceived quality, we use MPlayer-specific quantity *audio-video delay* (A-V) which indicates the difference in time between the end of the audio and the end of the video display for a given frame. MPlayer gives a warning that the processor is too slow for the video when the delay is greater than 0.5 seconds, and thus in the analysis below we consider this as a threshold that should not be reached.

Figures 2 through 5 present the impact of the CPU frequency on the A-V delay for the videos described in Figure 1. As shown by these figures, the behaviors fall into three categories: 1) At the higher frequencies, there is no or negligible delay. Any overrun due to a complex frame is quickly amortized by the slack time in the treatment of subsequent simpler frames. For videos encoded at a high resolution, such as Madagascar, this behavior is only achieved at the highest frequencies, while for videos encoded at a low resolution, such as X-Men 3, this behavior is possible at as little as 800 MHz. 2) As the frequency decreases, the computation time increases and there is more overrun and less slack time. An overrun is not amortized by the next few frames and the delay begins to accumulate, eventually reaching the 0.5 second threshold. Nevertheless, the computational requirements of frames vary greatly and, as shown by the case of Madagascar at 1200 MHz or Jarhead at 800 MHz, the delay eventually returns to an acceptable level. 3) When the frequency is too low to support the requirements of the movie, overruns are never amortized. As illustrated by Jarhead and Harry Potter at 600 MHz, the delay increases linearly until the audio runs out, and then falls off sharply as the player displays the rest of the frames as fast as possible.

The A-V delay gives us an externally defined metric against which to measure the quality of service, but is specific to MPlayer. Another, more generally applicable, perspective on the same information is the execution time for each frame. Figure 6 shows the execution times of frames 300-400 of Jarhead, which include first an action sequence and then a static title (the region circled in Figure 3). Just as the A-V delay shown in Figure 3 indicates that the video can be played with essentially no delay, the execution time shown here indicates that most of the frames are treated within the frame time of 41.7ms, and those that are not are quickly amortized by later ones. At 1000 MHz, the treatment of the frames in the action sequence always exceeds the frame time, but as shown in Figure 3, the accumulated overrun is not enough to cause an excessive delay. The situation changes at 800 MHz, where the treatment times are further above the frame time and the A-V correspondingly rises to unacceptable levels. Finally, at 600 MHz the treatment times of both the action sequence and the static title are far above the frame time, and the delay rises correspondingly. We furthermore observe that the execution time is quite stable, with an average variance of at most 2ms over 30 runs of the video, as shown by the right y axes in Figure 6.

3.2 The effect of frequency adaptation on battery lifetime

To be useful, a power management strategy for a single machine component must give an overall energy savings for the computation, taking into account all of the relevant components, such as the disk, the memory, the screen, etc. To measure the impact of frequency adaptation on energy consumption, we measure the time required to discharge a fully charged 1600 mWh battery while playing a video. We have used a rather old battery to reduce the benchmarking time. While the absolute lifetime depends on both the computational requirements of the application and the degree of wear on the battery, this approach measures directly the actual experience that a user could have in practice. Figure 7 presents the battery lifetime when playing the Jarhead pre-

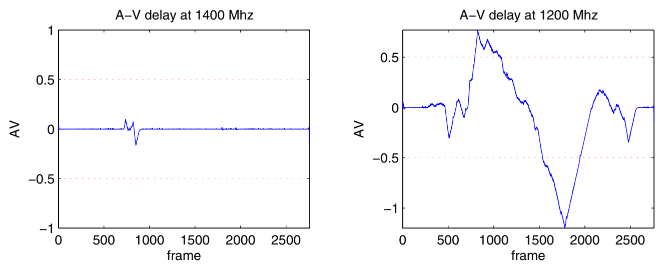


Figure 2: A-V delay for Madagascar

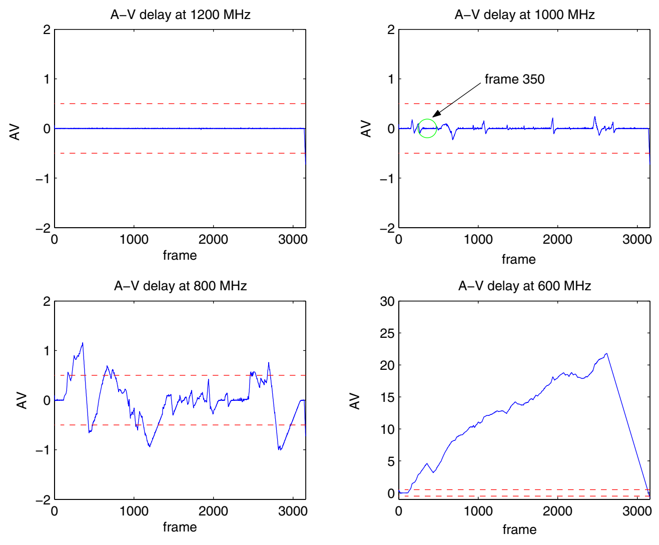


Figure 3: A-V delay for Jarhead

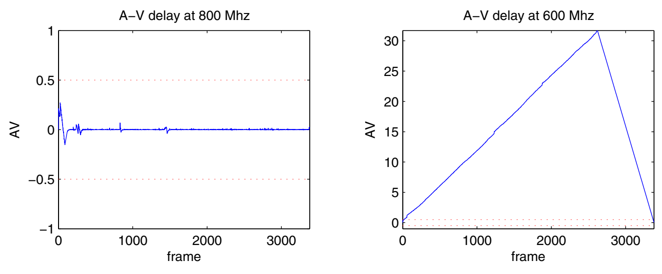


Figure 4: A-V delay for Harry Potter

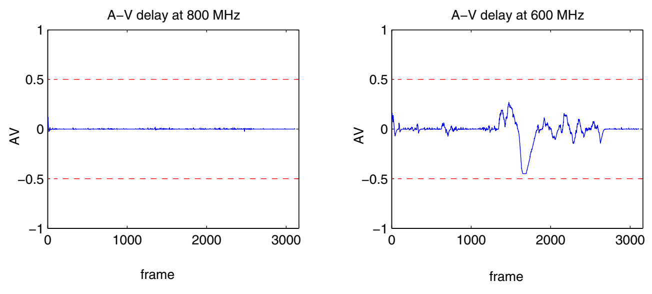


Figure 5: A-V delay for X-Men 3

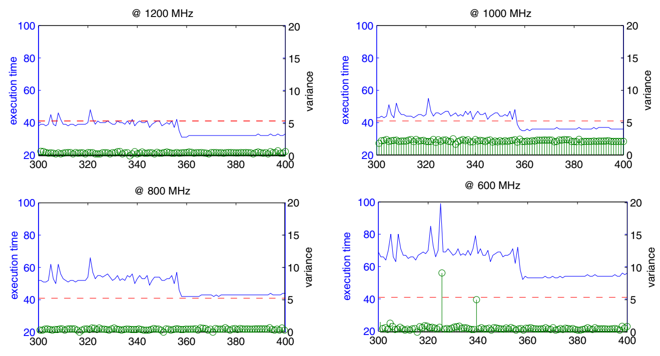


Figure 6: Frame time and variance for Jarhead

Static frequency	Battery lifetime
1700	18.2 minutes
1400	24.0 minutes
1200	25.0 minutes
1000	26.2 minutes

Figure 7: Battery lifetime when playing Jarhead

view at the frequencies that give acceptable video quality. Playing the video at 1000 MHz increases the battery lifetime by 44% as compared to running the CPU at full speed.

3.3 Assessment

Our experiments show that in the context of divx video playback there is a significant opportunity for reducing power consumption by scaling the CPU frequency. Divx videos exhibit a high variability in computational requirements between frames and across different videos, and thus existing power management strategies are not well suited to this setting. For example, Figure 8 shows the frequencies chosen by `powernowd` (version 0.96, as distributed with Ubuntu). `Powernowd` most often selects 1700 MHz for this video even though our measurements in Figure 3 show that the entire video can run at 1000 MHz with no perceptible delay. Furthermore, Figure 7 shows that using 1000 MHz rather than 1700 MHz entails a reduction of 44% in energy consumption.

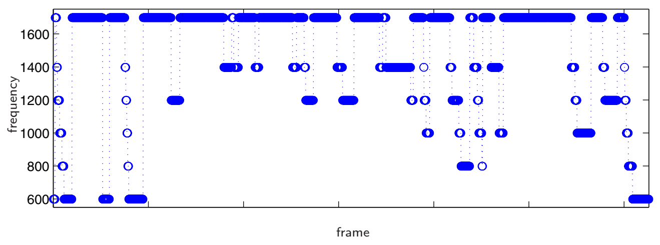


Figure 8: The frequencies chosen by `powernowd` when playing Jarhead

As compared to ordinary video display, however, the context of a video kiosk provides an additional source of information: the behavior of the video on previous iterations. Our measurements show that in contrast to previously used metrics, this metric is quite stable. Thus, we propose a solution, HbDVS, in which the CPU frequency is chosen based on a stored history of the previous playback of a video.

4. HISTORY-BASED DVS

Our approach, HbDVS, treats the video in two phases: an *adaptation phase* and a *post-adaptation phase*. The adaptation phase is used in the first few iterations of the video and creates a *frequency plan*, containing a frequency for each frame. The post-adaptation phase is used in all subsequent iterations of the video and treats each frame of the video at the CPU frequency indicated in the frequency plan.

4.1 Adaptation phase

The goal of the adaptation phase is to select a CPU frequency for each frame that is as low as possible while maintaining the video's timing requirements. In this it uses two modules: an optimistic *frequency selection module* and a pessimistic *feedback module*. The frequency selection module assigns each frame the frequency *just below* the lowest one where the player meets its frame rate, optimistically assuming that subsequent frames will absorb the induced overrun. The feedback modules detects situations where the overrun has not been absorbed and increases the frequency for some of the frames causing the overrun, pessimistically assuming that otherwise it will recur on subsequent iterations.

4.1.1 Frequency selection

The frequency selection module repeatedly runs the video iterating over the possible frequencies, from highest to lowest. On each iteration, it identifies the frames that must be treated at the current frequency to satisfy the video's timing requirements. This module uses the following concepts:

F_Master: the CPU frequency associated with the current iteration.

frequency_plan$_f$: the CPU frequency assigned to frame f (0 if no frequency has been assigned).

frame_time: the amount of time available for the treatment of each frame, *i.e.* the inverse of the frame rate.

ET_f: the treatment time for frame f.

δ: the expected variance in the treatment time (cf. Fig. 6).

overrun: the accumulated treatment time beyond the *frame_time* for the previous frames.

Within a iteration at frequency *F_Master*, the frequency selection module does the following for each frame f:

- Before treating the frame, the frequency selection module sets the CPU frequency to *frequency_plan*$_f$, if a frequency has been assigned for f, and *F_Master* otherwise.
- After treating the frame, the frequency selection module checks whether the frame should be assigned the frequency *F_Master* and updates the overrun. The frame is assigned *F_Master* if it has not already been assigned a frequency and if the following holds:

  $$ET_f + \delta + \mathit{overrun} > \mathit{frame\_time}$$

  The new overrun is computed as follows:

  $$\mathit{overrun} = \mathsf{max}(0, \mathit{overrun} + (ET_f - \mathit{frame\_time}))$$

  We do not record a negative overrun, as the player should sleep in this case. Furthermore, the overrun does not contain the variance, as the overrun is a measure over multiple frames, and the variance at each frame is thus assumed to cancel out.

frm	ET	freq	over-run
1	45	1000	4
2	38	1000	1
3	38	0	0
4	25	0	0

Iteration 1: *F_Master* = 1000MHz

frm	ET	freq	over-run
1	45	*1000*	4
2	38	*1000*	1
3	41	800	1
4	30	0	0

Iteration 2: *F_Master* = 800MHz

frm	ET	freq	over-run
1	45	*1000*	4
2	38	*1000*	1
3	41	*800*	1
4	34	600	0

Iteration 3: *F_Master* = 600MHz

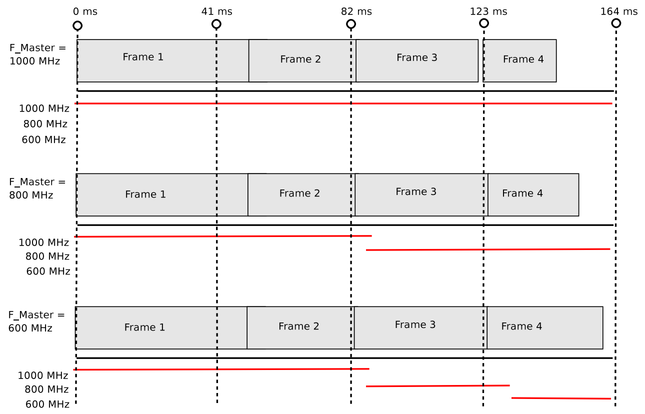


Figure 9: A simple example of frequency selection. Execution time (ET) is in ms. A frequency in italics is one that is obtained from the frequency plan.

At the end of a iteration at frequency *F_Master*, all of the frames that cannot be treated before the end of the frame time have been assigned a frequency, either *F_Master* in the current iteration or some higher frequency in a previous one.

Example. We illustrate frequency selection with the following example. Consider a processor with frequencies 1000 MHz, 800 MHz and 600 MHz, and a video of four frames with *frame_time* = 41ms. For simplicity, we assume that the variance is 0. Figure 9 shows a trace for this example.

When the video is played for the first time, *F_Master* is 1000MHz and *overrun* is initially 0. The treatment of frame 1 takes 45ms. This exceeds the *frame_time*, and thus the frame is assigned the frequency 1000MHz and *overrun* is set to $45 - 41$, or 4. The treatment of frame 2 takes only 38ms. Adding in *overrun*, we obtain 42ms, which exceeds *frame_time*. This frame is thus also assigned the frequency 1000MHz and *overrun* is set to $42-41$, or 1. The treatment of frame 3 again takes 38ms. Adding in *overrun*, we obtain 39ms, which is below the *frame_time*. Thus, *overrun* is set to 0 and no frequency is assigned to the frame. Finally, the treatment time of frame 4 is below *frame_time* and there is no overrun, so no frequency is assigned to this frame.

When the video is played for the second time, *F_Master* is 800MHz and *overrun* is reset to 0. Frames 1 and 2 are each treated at their stored frequency. Treatment of frame 3 requires 41ms, and adding the overrun gives 42ms, so this frame is assigned the frequency 800MHz. The treatment time for frame 4 combined with the overrun remains below *frame_time* and so no frequency is assigned to it.

When the video is played for the third time, *F_Master* is 600MHz and *overrun* is reset to 0. The first three frames are each treated at their stored frequency. The treatment time for frame 4 remains below *frame_time*, but there is no lower frequency, so this frame is assigned the frequency 600MHz.

4.1.2 Feedback

The feedback module is triggered when *overrun* exceeds a quality threshold, which we take to be the frame time.[1] The goal of this module is twofold: 1) to ensure that the degree of overrun is not repeated on subsequent iterations of the video, and 2) to reduce the overrun in the current iteration.

Before treating each frame, the feedback module checks whether the overrun accumulated by the treatment of the preceding frames exceeds the quality threshold. If this occurs at some frame f, it means that previous frames have been treated at a frequency that is too low for their combined computational requirements. To ensure that the problem does not repeat on subsequent iterations, we increase the frequency for some or all of these previous frames, as it is an invariant of the algorithm that the player was able to maintain the frame rate for these frames at all frequencies higher than the assigned ones. To restore the frame rate of the current iteration, the feedback module additionally increments the frequency for subsequent frames by one level, for the current iteration only, until the overrun is absorbed.

A key issue is the choice of which of the previous frames should have their frequency increased for subsequent iterations and by how much. To choose the frames for which to increase the frequency, we observe that increasing the frequency used for a frame gives maximum benefit if there is more accumulated overrun than the slack time introduced by the increase, so that all of the introduced slack time is used to absorb the overrun. This is most likely to be the case for the frame just before the frame f at which the overrun was observed to exceed the frame time. Thus, we first increment the frequency for this frame and work backwards from there, stopping at the first frame for which the overrun is 0, as incrementing the frequency for that frame will only cause the player to sleep. To determine by how much to increase the frequency, we assume that increasing the frequency for the overrun frames will give the same benefit in the subsequent iterations as increasing the frequency for the frames after f gives in the current iteration. This assumption is clearly an approximation, as changing frequency levels does not always have a uniform effect. If insufficient frames are adjusted, an overrun will be detected and accounted for on subsequent iterations. The algorithm provides no check whether too many frames are adjusted, however, we have found that fairly few frames are affected by the feedback module, and that such a situation would have very little impact on the overall power consumption in practice.

Finally, a remaining issue is the initialization of the frequency plan for the frames following f. Until the overrun is absorbed, such frames are treated at a frequency one level higher than the stored frequency, if available, or one level higher than *F_Master*, otherwise. This implies that a frame that does not have a stored frequency is not tested at *F_Master*. We do not assign a frequency to such frames and retain the same value of *F_Master* on the next iteration. This implies that the adaptation phase can consist of more iterations than there are frequencies, but we have observed that it reaches a fixed point quickly in practice.

[1]Note that this quality threshold is much more stringent than that of MPlayer, as a typical frame time of 41-42ms is less than 10% of MPlayer's A-V threshold of 0.5 seconds. We choose a more stringent threshold to ensure that the reduced power consumption does not come at the cost of playback quality.

Example. The behavior of the feedback module is illustrated by the example in Figure 10 for a video with a frame time of 41ms and a maximum frequency of 1000 MHz. On the first iteration, *F_Master* is 1000 MHz and all of the frames are treated within the frame time. On the second iteration, *F_Master* is 800 MHz. An overrun has accumulated in the part of the video preceding the frames shown in the example, and at the end of frame n the overrun has passed the frame time. Thus, the feedback module is triggered before the treatment of frame $n+1$. The frequency for frame n is increased by one level in the frequency plan, and the frequency for frame $n+1$ is set to one above *F_Master* for the current iteration. While this causes the overrun to decrease, it is not sufficiently absorbed and the feedback module is triggered again on frame $n+2$. This time, it is the frequency for frame $n-1$ that is increased by one level in the frequency plan, working backwards towards the start of the overrun. Frame $n+2$ is also run at the frequency above *F_Master* for the current iteration, which causes the overrun to go below the frame time. Finally, the third iteration uses *F_Master* as 800 MHz again, because frames $n+1$ and $n+2$ have not been tested at that frequency. This time, frames $n-1$ and n are run at 1000 MHz, according to the updated frequency plan. Overall, the overrun remains below the frame time within these frames on this iteration.

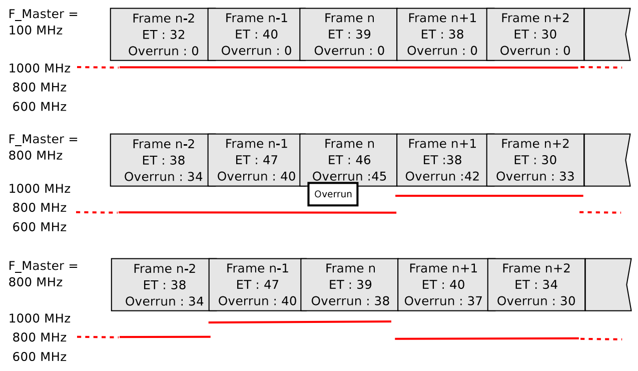

F_Master = 100 MHz
Frame n-2 ET : 32 Overrun : 0
Frame n-1 ET : 40 Overrun : 0
Frame n ET : 39 Overrun : 0
Frame n+1 ET : 38 Overrun : 0
Frame n+2 ET : 30 Overrun : 0
1000 MHz
800 MHz
600 MHz
F_Master = 800 MHz
Frame n-2 ET : 38 Overrun : 34
Frame n-1 ET : 47 Overrun : 40
Frame n ET : 46 Overrun :45
Frame n+1 ET :38 Overrun :42
Frame n+2 ET : 30 Overrun : 33
Overrun
1000 MHz
800 MHz
600 MHz
F_Master = 800 MHz
Frame n-2 ET : 38 Overrun : 34
Frame n-1 ET : 47 Overrun : 40
Frame n ET : 39 Overrun : 38
Frame n+1 ET : 40 Overrun : 37
Frame n+2 ET : 34 Overrun : 30
1000 MHz
800 MHz
600 MHz

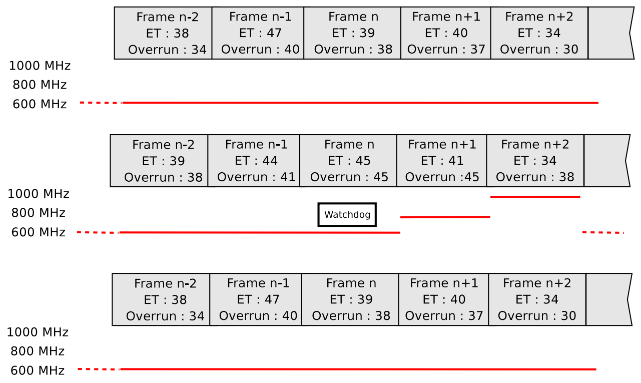
Figure 10: Feedback example

4.2 Post-adaptation phase

After the adaptation phase completes, the variance implies that it is possible, although unlikely, that a sequence of frames will accumulate a delay that exceeds the quality threshold. As the adaptation phase has ensured that the video can normally be displayed according to the frequency plan with acceptable quality, we do not make further modifications to the frequency plan in the post-adaption phase. Nevertheless, this phase includes a *watchdog* that detects such overruns and treats subsequent frames at higher frequencies within the current iteration until the delay has returned below the quality threshold.

The goal of the watchdog is to absorb the overrun as quickly as possible while minimizing the extra power consumption. Accordingly, subsequent frames are run at increasingly high increments above their stored frequency, until the overrun returns below the frame time. Specifically, the n^{th} frame f after the overrun was first observed is treated at the n^{th} frequency above $frequency_f$, up to the maximum frequency available on the machine.

This strategy is illustrated in Figure 11, again for a video with a frame time of 41ms. Although all frames are treated within an acceptable amount of time at 600 MHz in the first iteration, in the second iteration the overrun exceeds the frame time at the end of frame n. In this case, the watchdog is triggered first at frame $n+1$, and then, because the overrun is not sufficiently absorbed, it is triggered again at frame $n+2$. For frame $n+1$, a frequency one level higher than the stored frequency is used, while for frame $n+2$, a frequency two levels higher is used. At this point, the overrun goes below the frame time, and subsequent frames are again treated at their stored frequency. Finally, on the third iteration the frames are all treated at 600 MHz, as the increases to the frequency in the second iteration were for that iteration only, and have no effect on the frequency plan.



Frame n-2 ET : 38 Overrun : 34
Frame n-1 ET : 47 Overrun : 40
Frame n ET : 39 Overrun : 38
Frame n+1 ET : 40 Overrun : 37
Frame n+2 ET : 34 Overrun : 30
1000 MHz
800 MHz
600 MHz
Frame n-2 ET : 39 Overrun : 38
Frame n-1 ET : 44 Overrun : 41
Frame n ET : 45 Overrun : 45
Frame n+1 ET : 41 Overrun :45
Frame n+2 ET : 34 Overrun : 38
1000 MHz
800 MHz
600 MHz
Watchdog
Frame n-2 ET : 38 Overrun : 34
Frame n-1 ET : 47 Overrun : 40
Frame n ET : 39 Overrun : 38
Frame n+1 ET : 40 Overrun : 37
Frame n+2 ET : 34 Overrun : 30
1000 MHz
800 MHz
600 MHz

Figure 11: Watchdog example

4.3 Implementation

Our approach is implemented in MPlayer as a library providing the functions `init_dvs`, `first_frame_dvs`, `start_frame_dvs`, and `end_frame_dvs`, which behave as follows:

- `init_dvs`: This function initializes the various structures used by the algorithm, including setting *F_Master* to the highest frequency and the elements of the frequency plan to 0.
- `first_frame_dvs`: This function resets the various structures used by the algorithm at the start of a new iteration of the video. In particular, during the adaptation phase, *F_Master* is set to the next lower frequency if all of the unassigned frames have been tried at the current value of *F_Master* in the preceding iteration.
- `start_frame_dvs`: In the adaptation phase, this function executes the feedback module, and in the post-adaptation phase, it executes the watchdog, both based on the behavior of the previous frame. This function then uses the Cpufreq `userspace` governor [2] to set the CPU frequency to the one chosen for the current frame, if the CPU is not already at that frequency.
- `end_frame_dvs`: During the adaptation phase, this function executes the frequency selection module, which uses the treatment time of the frame to decide whether the frame should be assigned the frequency *F_Master*. This function does nothing in the post-adaptation phase.

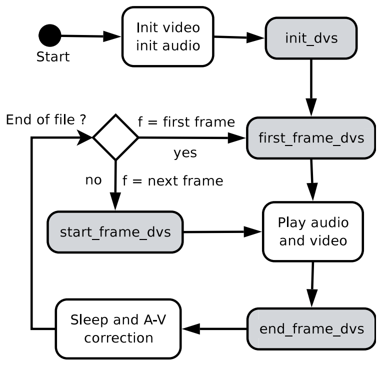


Figure 12: MPlayer architecture. Introduced function calls are shown in grey.

These functions amount to around 200 lines of code. The only change to the existing MPlayer source code is to add calls to these functions during initialization and before and after the treatment of each frame, as illustrated in Figure 12. There is no modification to the OS or to the video codec.

5. EVALUATION

We measure various properties of the video display when using HbDVS. All of the tests are carried out on the Intel Pentium 4M architecture described in Section 3.

Energy consumption. Figure 13 presents the power consumption for one iteration of each video. In the case of HbDVS, we use an iteration from the post-adaptation phase, in which the frequency plan has already been created. Measurements are taken according to the strategy used by Bellosa [1]. An ATMIO-16 E10 card is connected to the power supply of the Dell Inspiron laptop, and is used to measure the power consumption at a rate of 1000 Hz. In each case, the total power consumption with HbDVS is less than the power consumption at the minimum static frequency at which the video can be displayed correctly.

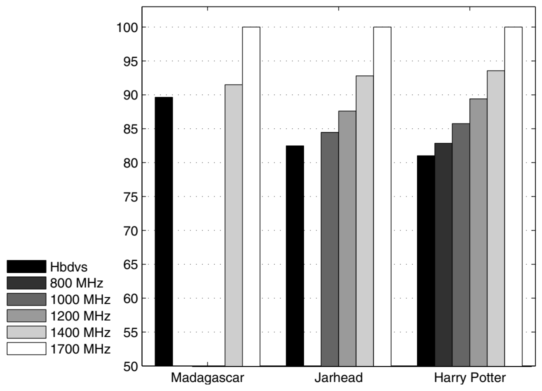


Figure 13: Power consumption for one iteration normalized to 1700 MHz

In practice, however, it is the battery lifetime that is important to the user, and this quantity is only indirectly to related to the measured power consumption. Figure 14 illustrates the effect of our algorithm on battery lifetime. We

Video	Policy	Battery lifetime (min)	HbDVS gain over other approaches	Power consumption for one iteration in the post-adaptation phase (joules)
Madagascar	1700 Mhz	18.5	1.30	3254
	1400 Mhz	17.2	1.40	2977
	powernowd	17.2	1.40	-
	HbDVS & adaptation	21.4	1.12	-
	HbDVS	24.0	1.00	2916
Jarhead	1700 MHz	18.2	1.54	3763
	1000 MHz	26.2	1.07	3177
	powernowd	22.0	1.27	-
	HbDVS & adaptation	26.2	1.07	-
	HbDVS	28.0	1.00	3103
Harry Potter	1700 MHz	21.0	1.48	3560
	800 MHz	25.0	1.24	2950
	powernowd	25.6	1.21	-
	HbDVS & adaptation	29.0	1.07	-
	HbDVS	31.0	1.00	2884
X-Men 3	1700 Mhz	16.3	2.09	-
	600 Mhz	34.5	0.99	-
	powernowd	34.0	1.00	-
	HbDVS & adaptation	32.5	1.05	-
	HbDVS	34.0	1.00	-

Figure 14: Battery lifetime and Power consumption. "HbDVS & adaptation" includes both the adaptation phase and the post-adaptation phase, while "HbDVS" refers to our approach using a previously computed frequency plan (the post-adaptation phase).

obtain an improvement of up to 109% as compared to the maximum frequency of the machine, up to 40% as compared to the minimum fixed frequency at which the entire video can be displayed with no perceptible loss of quality (see Figures 2 through 5), and 40% as compared to `powernowd`. Indeed, the only case where we obtain no improvement as compared to the minimum fixed frequency and `powernowd` is X-Men 3, which can run at the lowest frequency with no perceptible loss of quality (see Figure 5). We conjecture that if lower frequencies were available on the Intel Pentium 4M, our algorithm would take advantage of them, and we could further improve the battery lifetime in this case.

The frequency plan. Figure 15 shows the frequencies selected by our algorithm for each video and Figure 16 summarizes the percentage of frames treated above, at, and below the minimum fixed frequency. For a given video, our algorithm assigns up to 93% of the frames a lower frequency than the minimum fixed frequency at which the entire video can be displayed with no perceptible loss of quality.

Our algorithm is very fine-grained, in that it considers a single frame at a time, unlike `powernowd` that considers the load incurred by all of the frames within a give time interval. As a result, the frequency plan contains many changes in frequency, as illustrated in Figure 17 for Madagascar. The graph for Jarhead (not shown), is similar. For Harry Potter, there is frequent alternation between the frequencies 1000 MHz, 800 MHz, and 600 MHz, as shown in Figure 18. For X-Men 3 the frequency is essentially constant at 600 MHz. According to the Intel documentation on the Pentium M architecture [6], changing the frequency on this architecture incurs a delay of several tens of microseconds. Despite the many changes in frequency shown in our figures, they occur at most once per frame, which for our examples amounts to at most once every 33.37 milliseconds. Even with a frame rate of 100 frames per second, frequency changes would occur at most every 10 milliseconds, which is 1000 times the delay incurred by the frequency change. Thus, the overhead incurred by changing the frequency is negligible in our case.

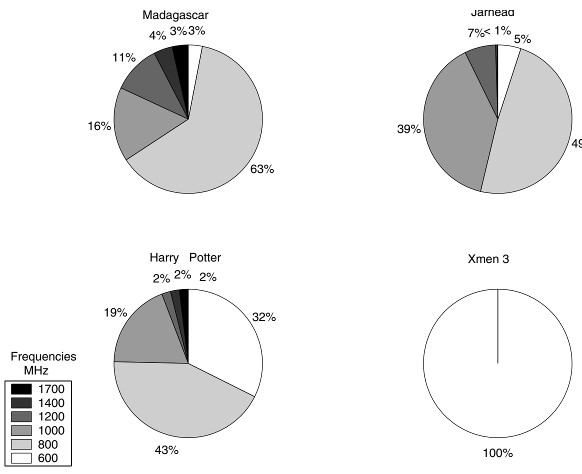


Figure 15: Frames at each frequency in the frequency plan

Video	mff.	frames > mff.	frames = mff.	frames < mff.
Madagascar	1400 MHz	3%	4%	93%
Jarhead	1000 MHz	7%	39 %	54%
Harry Potter	800MHz	25%	43%	32%
X-Men 3	600 MHz	0%	100%	0%

Figure 16: Comparison between the frequency plan and the minimum fixed frequency (mff.)

The impact of feedback. The frequency actually used on each iteration is determined by both the frequency plan and either the feedback module or the watchdog, depending on whether the iteration is part of the adaptation phase or the post-adaptation phase. Figure 19 shows the number of times the feedback module is triggered in each iteration of the adaptation phase. The feedback module is frequently triggered when *F_Master* first drops to a lower frequency. In this case, a sequence of frames that is treated at just under

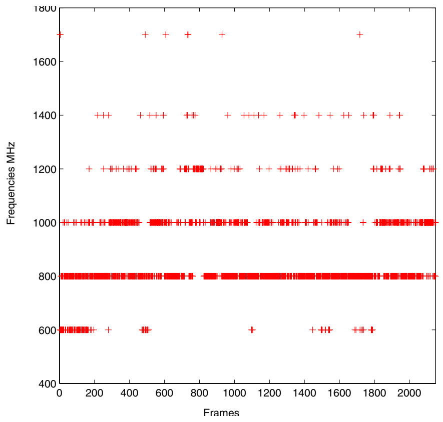


Figure 17: Frequency plan for Madagascar

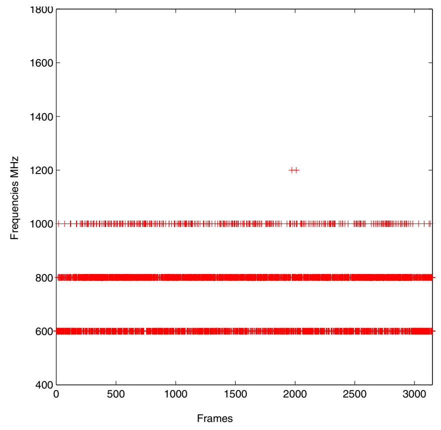


Figure 18: Frequency plan for Harry Potter

the frame time when using the higher frequency is treated at just over the frame time at the new frequency *F_Master*, eventually exceeding the quality threshold. In this case, the feedback module increases the frequency of a few of the preceding frames. The measurements show that this strategy is effective, as on subsequent iterations the feedback module is triggered quite rarely, and reaches a point where the overrun remains below the quality threshold within a few iterations. During the post-adaptation phase, the watchdog was never triggered in our experiments, showing that the frequency plan as calculated during the adaptation phase is adequate for the video.

The perceptible quality. Our algorithm is designed in terms of the execution time for each frame, while MPlayer measures quality in terms of the MPlayer-specific A-V delay. Figure 20 shows that our use of a quality threshold of one frame time keeps the A-V delay close to 0, with very little variation in the case of the higher resolution videos Madagascar and Jarhead and nearly no variation in the case of the lower resolution videos Harry Potter and X-Men 3. In all cases, the A-V delay is well within the MPlayer quality threshold of ±0.5.

Space consumption. Our approach requires maintaining a frequency plan for all videos in the set of videos currently

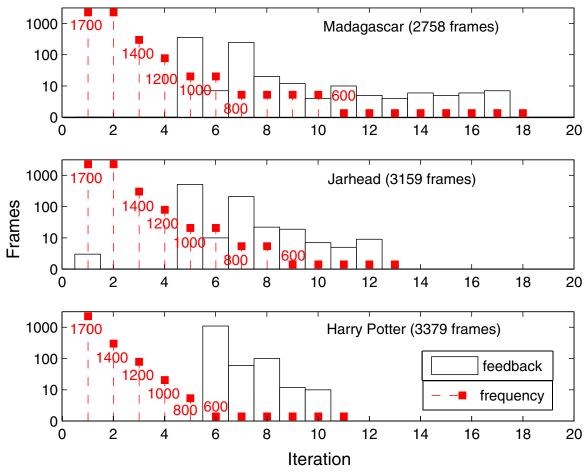


Figure 19: The number of frames on which the feedback module is triggered at various frequencies in the adaptation phase

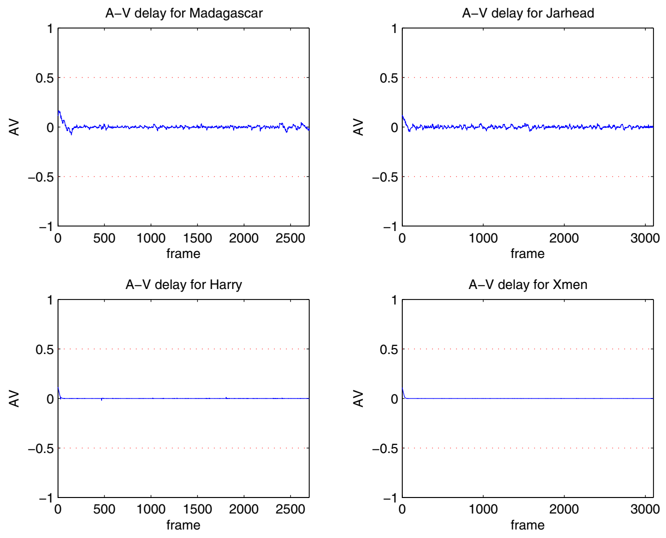


Figure 20: A-V delay

displayed by the kiosk. This frequency plan has size proportional to the total number of frames in these videos. Each entry of the frequency plan stores only an indication of the frequency assigned to the corresponding frame, or 0 if no frequency has yet been assigned. Four bits per frame are thus sufficient for a machine that provides up to 15 CPU frequencies. With this coding strategy, 450KB is sufficient to maintain the frequency plan for 10 hours worth of distinct video frames, encoded at 25 frames per second.

6. CONCLUSION AND FUTURE WORK

Video display is an attractive target for DVS because it has easily identifiable deadlines and there is often slack time available to absorb the additional computation time incurred by lowering the CPU frequency. Nevertheless, because of the high variability in the computational requirements of the various frames, previous mechanisms either are not highly effective on video, or have resorted to modifying the decoding algorithm or the OS. In this paper, we have shown how by exploiting a specific property of one kind of video display, the repetitive display of the same set of videos as found in kiosks, we can obtain an approach that is lightweight to implement, but gives results that are closely tailored to the video's computational requirements. In practice, our approach gives substantial improvement in battery lifetimes, up to 109% as compared to the maximum frequency on our test machine, up to 40% as compared to playing the video at the minimum fixed frequency that gives acceptable results for the entire video, and up to 40% as compared to the Linux tool `powernowd`.

We envisage several avenues for future work. In this work, we have considered an Intel Pentium 4M processor and the video player MPlayer. Preliminary results on an Intel Centrino with frequencies 600, 800, 900, 1000, 1100, 1200, 1300, and 1400 MHz are comparable to our results here. Nevertheless, we would like to study the approach on a wider variety of architectures, and with other video players. As presented here, HbDVS always starts the adaptation phase by treating every frame at the maximum frequency. Another approach would be to start from a frequency plan created for another, similar, machine. This approach would be particularly useful if the video is to be displayed only a few times, but would still allow the energy usage to adapt to the precise requirements of the host machine. Finally, we are considering refinements to the feedback strategy, both to consider the effect of relaxing the quality threshold to allow an occasional delay of more than one frame and to identify cases where an overrun is likely to repeat on the next iteration and to augment the frequency more aggressively in these cases.

Availability. The implementation of our algorithm is available at `http://www.emn.fr/x-info/rurunuel/hbDVS.html`

7. REFERENCES

[1] F. Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Science, June 2001.

[2] D. Brodowski. Linux kernel CPUfreq subsystem. http://www.kernel.org/pub/linux/utils /kernel/cpufreq/cpufreq.html.

[3] L.-O. Burchard and P. Altenbernd. Estimating decoding times of MPEG-2 video streams. In *Proceedings of International Conference on Image Processing (ICIP 00)*, Vancouver, Canada, Sept. 2000.

[4] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, Apr. 1992.

[5] K. Flautner and T. N. Mudge. Vertigo: Automatic performance-setting for Linux. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–116, Boston, MA, Dec. 2002.

[6] D. Genossar and N. Shamir. Intel Pentium M processor power estimation, budgeting, optimization and validation. *Intel Technology Journal*, 7(2):44–49, May 2003.

[7] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First Annual International Conference on Mobile Computing and Networking (MOBICOM '95)*, pages 13–25, Berkeley, CA, Nov. 1995.

[8] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 73–86, San Diego, CA, Oct. 2000.

[9] C. Im and S. Ha. Dynamic voltage scaling for real-time multi-task scheduling using buffers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 88–94, Washington, DC, USA, July 2004.

[10] Intel. *Intel SpeedStep Technology*, Jan. 2000.

[11] J. R. Lorch and A. J. Smith. PACE: A new approach to dynamic voltage scaling. *IEEE Trans. Computers*, 53(7):856–869, 2004.

[12] A. Maxiaguine, S. Chakraborty, and L. Thiele. DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 111–116, New York, NY, USA, 2005. ACM Press.

[13] T. Pering and R. Broderson. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design, 1998*, pages 76–81, Monterey, CA, June 1998.

[14] P. Pillai and K. G. Shin. Real-Time dynamic voltage scaling for Low-Power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 89–102, Banff, Canada, Oct. 2001.

[15] J. Pouwelse. *Power Management for Portable Devices.* PhD thesis, Delft University of Technology, Oct. 2003.

[16] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3):195–237, 2005.

[17] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 13–24, Berkeley, CA, USA, Nov. 1994. USENIX Association.

[18] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES'02*, pages 238–246, Grenoble, France, Oct. 2002.

[19] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 149–163, Bolton Landing (Lake George), New York, Oct. 2003.