# DVS for Buffer-Constrained Architectures with Predictable QoS-Energy Tradeoffs

Alexander Maxiaguine[1] Samarjit Chakraborty[2] Lothar Thiele[1]
[1]Computer Engineering and Networks Laboratory, ETH Zürich
[2]Department of Computer Science, National University of Singapore
E-mail: {maxiagui,thiele}@tik.ee.ethz.ch, samarjit@comp.nus.edu.sg

## ABSTRACT

We present a new scheme for dynamic voltage and frequency scaling (DVS) for processing multimedia streams on architectures with restricted buffer sizes. The main advantage of our scheme over previously published DVS schemes is its ability to provide hard QoS guarantees while still achieving considerable energy savings. Our scheme can handle workloads characterized by both, the data-dependent variability in the execution time of multimedia tasks and the burstiness in the on-chip traffic arising out of multimedia processing. Many previous DVS algorithms capable of handling such workloads rely on control-theoretic feedback mechanisms or prediction schemes based on probabilistic techniques. Usually it is difficult to provide QoS guarantees with such schemes. In contrast, our scheme relies on worst-case interval-based characterization of the workload. The main novelty of our scheme is a combination of offline analysis and runtime monitoring to obtain worst case bounds on the workload and then improving these bounds at runtime. Our scheme is fully scalable and has a bounded application-independent runtime overhead.

## Categories and Subject Descriptors

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

## General Terms

Algorithms, Performance, Design

## Keywords

DVS, Buffer management, QoS, Predictable design

## 1. INTRODUCTION

Multimedia applications typically exhibit a high degree of variability in the workload generated by them. Recently, a number of advanced Dynamic Voltage and Frequency Scaling (DVS) techniques have used buffers for smoothing out this variability, with the aim of saving the energy consumed by a processor [3–8]. These techniques can be broadly classified into three groups based on how they perform the *workload prediction* required for DVS. [3, 7] predict the future workload based on stochastic models. [6, 8] employ feedback control loops to track workload changes and extrapolate the future workload. [4, 5] rely on offline worst-case characterization of tasks and statically use this characterization at runtime for making conservative scheduling decisions (i.e. essentially they do not use *prediction* as such, but assume that at any time the worst case workload will occur).

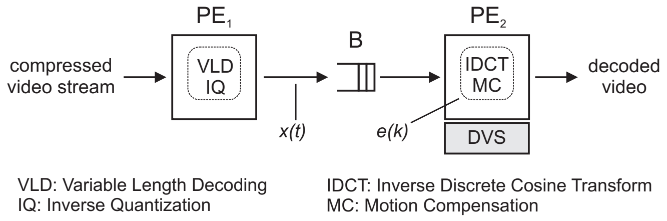

**Figure 1: MPEG-2 decoder implemented on two PEs**

Although it has been shown that the above lines of work lead to considerable energy savings, all of them still suffer from a number of drawbacks. Schemes based on stochastic prediction and feedback control are useful in handling workloads which are characterized by both, the data-dependent variability in the execution time of multimedia tasks and the burstiness in the on-chip traffic arising out of multimedia processing. However, typically it is difficult to provide hard QoS guarantees with such schemes. On the other hand, schemes that rely on worst-case characterization of the workload can provide hard QoS guarantees. However, they account only for the task execution time variability, and often assume that tasks arrive periodically.

In this paper we present a DVS technique which addresses the above mentioned shortcomings of the previous approaches. It takes into account both, the burstiness in a stream and the data-dependent variability in the execution time of a task. Additionally, our scheme offers a guaranteed QoS along with energy savings that are comparable with those obtained by previous approaches. One of the main assumptions made by many existing DVS schemes has been the availability of large buffers. However, in reality, many portable devices have severe cost and memory constraints. We also address this issue by targeting our DVS scheme specifically towards *buffer-constrained* architectures.

Our scheme relies on an offline analysis to determine *bounds* on the variability in the workload associated with a *class* of multimedia streams. At runtime, by using a bounded-length history which records the actually incurred workload, the bounds obtained from the offline analysis are revised. Such revised bounds are then used to adjust the voltage/frequency of the processor. These bounds are much tighter than those that can be obtained by a purely offline analysis. At the same time they are "safe" in terms of guaranteeing QoS constraints. Although this general scheme is very intuitive, it is not at all clear what kinds of *bounds* can be subjected to runtime revisions. The main result in this paper is a formalization of such bounds and efficient algorithms for revising them at runtime. Although we present this scheme in the context of a simple setup, where DVS is implemented on a single processor running a multimedia task (see Figure 1), it can also be applied to more involved architectures such as on-chip networks and multiple clock domain processors.

## 2. MOTIVATING EXAMPLE

Figure 1 shows an architecture for decoding MPEG-2 video streams. The two processing elements $PE_1$ and $PE_2$ may be general-purpose processors or embedded processor cores specialized for video processing. A compressed video stream first enters $PE_1$, which executes a part of the MPEG-2 decoding algorithm. The task running on $PE_1$ performs VLD and IQ functions. After processing on $PE_1$, the video stream (which is a sequence of partially decoded *macroblocks*) enters buffer $B$ at the input of $PE_2$. $PE_2$ reads $B$ one macroblock at a time and applies the IDCT and MC functions. Finally, the fully decoded video stream emerges at the output of $PE_2$.

Our objective is to minimize the energy consumed by $PE_2$ without deteriorating the quality of the processed video stream. The stream's quality is preserved if buffer $B$ at the input of $PE_2$ never overflows and if the processing delay, experienced by the stream on $PE_2$, does not exceed some specified value. Our ultimate goal is to design a *predictable system*. This means that we want to ensure that the system satisfies the above mentioned QoS requirements under *all* possible load scenarios and not only in the average case.

We assume that $PE_2$ supports DVS, i.e. its clock rate and supply voltage can be changed at run time (we always assume that the processor's supply voltage is changed appropriately whenever its clock rate is changed). Such changes can be controlled by the software on $PE_2$ or by some other hardware or software entity, external to $PE_2$. We refer to time instants at which the processor speed is altered as *adaptation points*.

In this paper, we assume that the adaptation points are fixed in time. For obtaining energy savings while providing the QoS guarantees, such an assumption is less favorable than the assumption that we can adapt the processor's clock rate at any time. Our method can however handle both these cases. In any case, the spacing of the adaptation points in time in our method is completely decoupled from the execution state and granularity of the tasks on $PE_2$. This means that the adaptations are not restricted to occur at task boundaries, and their frequency, in general, is independent of the rate at which the video stream arrives at the input of $PE_2$.

A DVS scheduler can reduce the energy dissipated on $PE_2$ by exploiting the variability of the workload imposed on this PE. This variability comes from two sources. First, the execution time of the task running on $PE_2$ is variable. Second, the data-dependent variability in the execution time of the task running on $PE_1$ causes the stream of macroblocks at the input of $PE_2$ to be *bursty*.

A traditional way of reducing the energy dissipated on $PE_2$ would be to fully average out the workload imposed on it using buffer $B$. If buffer $B$ is sufficiently large, it can completely absorb the workload fluctuations. This allows $PE_2$ to run at a low *constant* clock rate which is just sufficient to sustain the long-term average arrival rate of the stream. In this mode, one can ensure that $B$ never gets empty and therefore no cycles are wasted during low-load periods. This strategy would yield the most energy savings on $PE_2$. However, such a strategy is often not affordable since it requires large buffers for processing bursty multimedia workloads like MPEG streams. As an example, our experiments showed that the complete averaging of the workload imposed by DVD-quality videos on $PE_2$ required in the worst case a buffer space of at least 8100 macroblocks (or about 3.7 MByte). Such a large buffer would be too expensive to implement in some embedded SoC architectures. Further, from the application perspective, the delay incurred by the video stream on $PE_2$ as a result of such averaging might not be tolerable. (It is about 5 full video frames in our setup.)

In contrast, we assume that our architecture is *buffer-constrained*, i.e. the buffer space at the input of $PE_2$ is inadequate for the complete workload averaging. Hence, unless we allow buffer overflows we cannot constantly run the processor at the average rate: a burst in the stream's arrival pattern or in its execution demand can easily cause an overflow. To avoid such overflows, we could service the stream at some constant *safe* rate which is high enough to successfully handle the bursts under the given buffer constraint. Clearly, such a safe rate would be higher than the average rate. Therefore, during periods with average or low load some processor cycles would be wasted due to waiting on the empty buffer. In some cases we could save some energy by putting the processor into a low-power idle state whenever the buffer is empty and then let it run again whenever there is something to process in the buffer, i.e. use Dynamic Power Management (DPM). However, in many cases the switching overhead between processor power states is too high (in the range of milliseconds) compared to the stream arrival rate (in the range of microseconds in our case) thereby making this strategy infeasible. On the other hand, a DVS strategy which could exploit the *slack* during low-load periods would yield higher energy savings.

By now it should be clear that in spite of the buffer constraint there is a potential to save energy by running $PE_2$ at a lower rate during low-load periods. However, this potential should be realized very carefully since a burst may suddenly arrive and cause a buffer overflow. Hence, our goals are conflicting: on one hand we want to stay at the average level of performance for saving the energy, but on the other hand we have to provide QoS guarantees and, therefore, need to be ready at any time to handle the worst case scenario. Designing a scheduling strategy which can meet both these goals is a challenging problem and involves delicate tradeoffs.

The main challenge in designing a safe DVS strategy for a system with constrained buffers, as the one described above, is in the fact that it is a priori unknown how the workload will behave in an interval between two adaptation points. Even if we knew exactly how many stream objects will arrive within the interval, this information would be insufficient to guarantee that the buffer will not overflow. For providing such a guarantee we need to know *how* the stream objects will arrive within the interval. For instance, they may arrive in a dense burst right in the beginning of the interval. If in the previous interval the processor has not cleared enough buffer space to accommodate this burst, an overflow is bound to happen. Many existing DVS techniques which are capable of providing QoS guarantees and which employ buffers for energy reduction avoid this problem by assuming that the stream arrives into (or departs from) the buffer at a constant rate. This assumption, however, greatly simplifies the problem and often does not hold in practice. Further, by making this assumption, the existing techniques lose the opportunity to gain additional energy savings by exploiting the variability in the arrival process of the stream. They can only exploit the slack resulting from the variability in the task execution time.

## 3. PROPOSED METHOD

Figure 2 shows an overview of our method. Our algorithm dynamically adapts the clock rate of the processor to the workload variation using two mechanisms: (i) run-time monitoring of the buffer fill level (i.e. buffer $B$ in Figure 1) and (ii) on-line improvement of static worst-case bounds based on a workload history. It exploits both types of workload variability – the slack in the execution time and the irregular arrival pattern of the stream.

Central to our method is the concept of *worst-case interval-based workload characterization*. This concept is the key to providing QoS guarantees and achieving good average performance. The concept relies on the fact that peak load periods of multimedia workloads are bounded in their length and rate. The relevant worst-

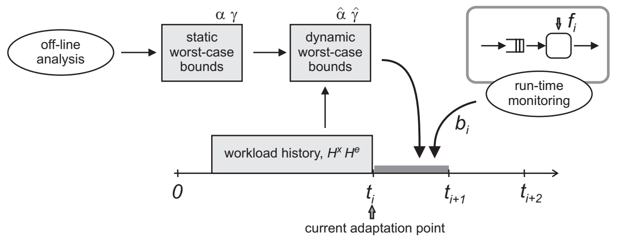

**Figure 2: Overview of the method**

case characteristics of the multimedia workloads are captured by worst-case bounds which we call *arrival and execution demand curves*. These curves represent a more detailed characterization of the workload compared to traditional task and event models. Using these bounds at runtime and taking into account the current backlog in the buffer and the workload history, our algorithm makes *safe, but not too pessimistic* decisions at the adaptation points.

We distinguish between two types of worst-case bounds: *static bounds* and *dynamic bounds* (see Figure 2). The static bounds are obtained at the design time using an offline analysis and are then used by the scheduler at runtime. In this sense they are similar to conventional workload characterizations such as worst-case execution times of tasks.

The use of dynamic bounds is a novel concept that we introduce in this paper. The novelty of this concept lies in the fact that these *worst-case* bounds are obtained *at run-time*. They represent an online improvement of the static bounds. The dynamic bounds are obtained using the workload history. Depending on the nature of the workload variation, these bounds can be much tighter than the corresponding static bounds. They allow our scheduler to be less pessimistic about the future workload and as a result achieve considerable energy savings. Further, the dynamic bounds provide the same level of guarantee as those provided by the static bounds from which they were derived. We will illustrate this concept using the following example.

EXAMPLE 1. *Suppose that the upper bound on the workload imposed by a stream on a processor within* any *time interval of length $\Delta$ equals to $A$ processor cycles. Since this bound holds at all intervals of length $\Delta$, it is a static worst-case bound. Now, suppose that we observe the system at some point in time $t$. If we know that the execution demand requested by the stream during time interval $[t - \Delta_p, t]$ (where $\Delta_p < \Delta$) was $B$ processor cycles, then we can guarantee that over the next time interval $(t, t + \Delta - \Delta_p]$, the stream will not request more than $A - B$ cycles. The value $A - B$ represents a dynamic worst-case bound over the interval $(t, t+\Delta-\Delta_p]$. At time $t$, the scheduler can safely use this dynamic worst-case bound for computing the frequency at which the processor needs to be run during the interval $(t, t + \Delta - \Delta_p]$.*

The following subsections give the details of this method.

## 3.1 Workload and service characterization

**Workload characterization:** We shall consider a stream to be composed of a potentially infinite sequence of *stream objects*. Depending on the application at hand a stream object can be an audio sample, a video (macro)block or a whole frame etc. A stream can be modeled using two functions $x(t)$ and $e(k)$ (see Figure 1). $x(t)$ denotes the total number of stream objects that arrived at the buffer $B$ during the time interval $[0, t]$, whereas $e(k)$ denotes the total number of execution cycles requested from the processor by $k$ consecutive stream objects starting from the first stream object in the sequence. Our objective is to characterize a whole *class* of streams that the processor has to handle. We achieve this by using an arrival curve $\alpha$ and an execution demand curve $\gamma$. These curves represent upper bounds, such that for any $i$ the following hold.

$$x_i(t + \Delta) - x_i(t) \leq \alpha(\Delta) \quad \forall t, \Delta \in \mathbb{R}_{\geq 0} \tag{1}$$

$$e_i(k + \epsilon) - e_i(k) \leq \gamma(\epsilon) \quad \forall k, \epsilon \in \mathbb{Z}_{>0} \tag{2}$$

$\alpha$ and $\gamma$ provide the worst-case interval-based characterization of the streams. $\alpha$ characterizes burstiness of stream arrivals on different interval lengths $\Delta$, while $\gamma$ characterizes variability in the execution demand. A tuple $(\alpha, \gamma)$ models a class of streams. In practice, such a class can encompass all streams belonging to a particular application scenario. For instance, all MPEG-2 video sequences with identical parameters like resolution, frame rate, bit rate etc. can belong to one class. $\alpha$ and $\gamma$ can be obtained analytically or from simulating a number of representative streams from the class. In the former case, our method provides hard performance guarantees.

**Service characterization:** The clock rate of a processor changes over time (due to DVS). Hence, we have to model this change properly. For this we use the concept of a *service curve* $\beta$, such that $\beta(\Delta)$ represents a lower bound on the amount of service offered to a stream within any interval $\Delta \geq 0$. This service can be measured in terms of processor cycles, i.e. $\beta(\Delta)$ denotes the number of processor cycles available within any time interval of length $\Delta$. Alternatively, $\beta(\Delta)$ can represent the number of stream objects that the processor guarantees to process within any $\Delta$. In the simplest case, when the processor constantly runs at a clock rate $f$, the service curve $\beta(\Delta) = f \cdot \Delta$. For a processor which may be switched off for maximum $\delta$ time units and in the rest of the time runs at a constant speed $f$ (e.g. as in the case of DPM), $\beta(\Delta) = \sup\{f \cdot (\Delta - \delta), 0\}$.

## 3.2 Theoretical background

The main property of our algorithm is its ability to provide QoS guarantees with respect to the delay and buffer constraints. Hence, we need to formally reason about the maximal delay and backlog which may be experienced by any stream while being processed by a processor. We base our formal reasoning on the results of *Network Calculus* [2]. It has been shown that upper bounds on the delay experienced by any stream object on the processor and the backlog in the buffer at the processor's input can be computed as:

$$\textit{delay} \leq \sup_{\forall \Delta \geq 0}\{\inf\{d \geq 0 : \alpha(\Delta) \leq \beta(\Delta + d)\}\} \tag{3}$$

$$\textit{backlog} \leq \sup_{\forall \Delta \geq 0}\{\alpha(\Delta) - \beta(\Delta)\} \tag{4}$$

The processor's rate is *safe*, if continuously running the processor at this rate guarantees that the buffer at its input never overflows and the delay constraint associated with the processed stream is satisfied. Our goal is to determine the *minimum* safe rate.

Suppose that all stream objects have identical execution demand. Then the service rate $R$ which the processor offers to a stream can be measured in terms of the number of stream objects processed per unit time. Thus, we can model the offered service as $\beta(\Delta) = R \cdot \Delta$. Then, from (4) we can find the minimum processor rate $R_L$ which ensures that the buffer of size $L$ never overflows: $R_L = \sup_{\forall \Delta \geq 0}\{(\alpha(\Delta) - L)/\Delta\}$

From (3) we can determine the minimum processor rate $R_D$ satisfying the delay constraint $D$: $R_D = \sup_{\forall \Delta \geq 0}\{\alpha(\Delta)/(D + \Delta)\}$ Thus, the minimum safe rate $R_{\mathit{safe}} = \max\{R_L, R_D\}$.

## 3.3 Adapting processor speed at run time

To save energy, during low-load periods our scheduler tries to run the processor at a rate which matches the stream's arrival rate. The scheduler uses the buffer fill level as an indicator of the stream's arrival rate. At each adaptation point, the scheduler tries to set the processor rate such that the buffer fill level is *close* to zero. Since any such rate tends to match the arrival rate and is lower than

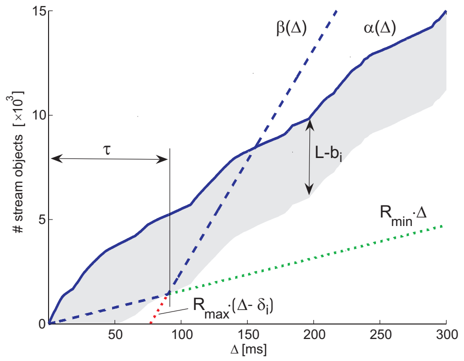

**Figure 3: Calculation of the minimum service rate $R_{min,i}$ for the $i$th adaptation interval.**

the minimum safe rate $R_{safe}$, this strategy results in energy savings during the low-load periods.

If a workload burst starts arriving, the processor frequency is increased accordingly. This is done in a safe way, based on the information about the current buffer fill level and the expected future worst-case workload. The scheduler tries to *fully* exploit the available buffer space during the bursts and be as "lazy" as possible. At each adaptation point, it sets the processor rate such that it is just sufficient to avoid a buffer overflow in the worst case. That is, if the worst case did really happen, the buffer would reach its full state but would not overflow at any time within the adaptation interval. Since the worst case happens rarely, this "lazy" strategy results in energy savings during the high-load periods.

Let $b_i$ denote the backlog in the buffer at $i$th adaptation point and suppose that the $i$th adaptation interval is of length $\tau$. Then the above rate-adaptation strategy can be realized if at the $i$th adaptation point the processor rate is set to $R_{L,i}$ which is computed as follows:

$$R_{L,i} = \sup_{0<\Delta\leq\tau} \{(\alpha(\Delta) - L + b_i)/\Delta\} \tag{5}$$

(5) ensures that the buffer will not overflow *within* the $i$th adaptation interval. However, it might happen that due to a burst the backlog at the end of $i$th adaptation interval is close to its maximum allowed value. In this case, avoiding buffer overflows may require the processor to run at a high rate during the next adaptation interval. This rate might be higher than the maximum rate $R_{max}$ supported by the processor. Thus, a deadlock situation may occur. To avoid such a situation, at $i$th adaptation point the scheduler has to also consider what might happen in the worst case *after* the $i$th adaptation interval. For this, at the $i$th adaptation point it sets the processor rate such that this rate is at least as high as $R_{min,i}$ which is computed as follows.

$$\delta_i = (L - b_i)/R_{max} + \inf_{\forall\Delta\geq\tau}\{\Delta - \alpha(\Delta)/R_{max}\} \tag{6}$$

$$R_{min,i} = R_{max}(\tau - \delta_i)/\tau \tag{7}$$

Figure 3 illustrates the above formulas. $R_{min,i}$ ensures that whenever the worst-case load *really* occurs in the $i$th interval and the buffer is (almost) full at its end, we can still prevent the buffer from overflowing. For this, after the $i$th adaptation interval, we just have to run the processor at $R_{max}$. This is because in the worst case the stream is guaranteed to receive a service which is not less than the service curve $\beta(\Delta) = \sup_{\forall\Delta>0}\{R_{min,i}\Delta, R_{max}(\Delta - \delta_i)\}$ and $\alpha(\Delta) - \beta(\Delta) \leq L - b_i$ for all $\Delta > 0$.

The opposite situation might also occur. If there is a lot of free space in the buffer, (5) and (7) may return zero. In this case, one could switch off the processor until the next adaptation point. However, this would not be an optimal energy saving strategy. Even if the stream's arrival rate is low during the time when the processor is switched off, a number of stream objects will accumulate in the buffer. This will necessitate the processor to run at a higher rate during the subsequent adaptation intervals. The optimal strategy is to run the processor at a rate $R^l_{min} = \mu(\tau)/\tau$, where $\mu(\tau)$ returns the minimum number of stream objects that might arrive within *any* interval of length $\tau$.

Finally, our algorithm can be summarized as follows:

$$R_i = \max\{R_{L,i}, R_{min,i}, R^l_{min}, R_D\} \tag{8}$$

where $R_i$ is the rate set by our algorithm at the $i$th adaptation point.

**Accounting for variable execution demand:** The above discussion was based on the assumption that all stream objects impose exactly the same execution demand on the processor. In reality this rarely happens. Assuming that the service rate measured in terms of stream object is constant, may lead to overly pessimistic results. To address this problem, our method employs the execution demand curve $\gamma$. $\gamma(\alpha(\Delta))$ gives an upper bound on the number of *cycles* that can be requested within any time interval of length $\Delta$ by any stream belonging to the class characterized by tuple $(\alpha, \gamma)$. For brevity and simplicity we skip the details of how exactly $\gamma$ is inserted in the above computations and present only the final result:

$$f_D = \sup_{\forall\Delta\geq0} \{\gamma(\alpha(\Delta))/(D + \Delta)\}$$

$$f_{L,i} = \sup_{0<\Delta\leq\tau} \{\gamma(\alpha(\Delta) - L + b_i)/\Delta\}$$

$$f_{min,i} = \sup_{\forall\Delta\geq\tau} \{f_{max}(\tau - \Delta) + \gamma(\alpha(\Delta) - L + b_i)\}/\tau$$

where $f_D$, $f_{L,i}$, $f_{min,i}$ and $f^l_{min}$ are processor rates in number of clock cycles corresponding to $R_D$, $R_{L,i}$, $R_{min,i}$ and $R^l_{min}$. Our algorithm therefore computes $f_i = \max\{f_{L,i}, f_{min,i}, f^l_{min}, f_D\}$.

## 3.4 Using dynamic worst-case bounds

The algorithm described in the previous subsection uses the static worst case bounds $\alpha$ and $\gamma$. It was derived under the assumption that at any point in time nothing is known about the past workload. Now suppose that we keep a finite-length workload history. By exploiting this history we can improve the energy savings without jeopardizing the safety property of the algorithm. We use the history to revise the static bounds $\alpha$ and $\gamma$ into their dynamic equivalents $\hat{\alpha}$ and $\hat{\gamma}$. This subsection explains how this is exactly done.

In Example 1, we employed a static bound which provided us the information about the worst-case workload only on intervals of length $\Delta$. We used this information as a *constraint* to compute the dynamic worst-case bound for a future interval of a smaller length. However, we can derive many such constraints from the curves $\alpha$ and $\gamma$, since they capture the worst-case workload on intervals of different lengths. We can then combine these constraints.

Suppose that the system is at the beginning of the $i$th adaptation interval of length $\tau$. The upper bound $\mathcal{X}_i$ on the number of stream objects that can arrive within $\tau$ is

$$\mathcal{X}_i(\tau) = \inf_{0\leq j\leq N}\{\alpha(j\theta + \tau) - H^x_i(j)\} \tag{9}$$

where $H^x_i$ is the *arrival history* at the $i$th adaptation point, $\theta$ is the *resolution of the arrival history*, and $N$ is the number of constraints that the scheduler considers for computing $\mathcal{X}_i$. $\theta$ can be interpreted as a sampling period with which arrivals are monitored.

The arrival history $H^x_i$ represents a set of $N$ sliding windows, with the $j$th window spanning the interval $[t_i - j\theta, t_i)$ and returning the number of stream objects that arrived within it. Formally,

$$H^x_i(j) = x(t_i) - x(t_i - j\theta), \quad j = 0, 1, .., N \tag{10}$$

| Videos | | | | Parameters |
|---|---|---|---|---|
| # | file name | # | file name | MP@ML |
| 1 | bbc3_080.m2v | 4 | susi_080.m2v | 8 Mbps CBR |
| 2 | cact_080.m2v | 5 | tens_080.m2v | 25 fps |
| 3 | mobl_080.m2v | | | 704×576 pixel |
| Source: `ftp.tek.com/tv/test/streams/Element/MPEG-Video/` | | | | |

**Table 1: MPEG-2 video sequences used in the experiments**

Note that from (1), (9) and (10) it follows that $\mathcal{X}_i(\tau) \leq \alpha(\tau)\ \forall i \in \mathbb{Z}_{\geq 0}, \forall \tau \in \mathbb{R}_{\geq 0}$. Hence, for the processor rate calculation, instead of using $\alpha(\tau)$ we can use $\mathcal{X}_i(\tau)$. Furthermore, since $\alpha$ is a wide-sense increasing function, i.e. $\Delta_1 > \Delta_2 \Rightarrow \alpha(\Delta_1) \geq \alpha(\Delta_2)$, $\mathcal{X}_i(\tau)$ can be used to improve $\alpha$ not only for $\tau$, but also for other interval lengths that are smaller than $\tau$:

$$\hat{\alpha}_i(\Delta) = \inf_{\forall \Delta \in [0,\tau]} \{\mathcal{X}_i(\tau), \alpha(\Delta)\} \tag{11}$$

(11) represents the *dynamic arrival curve* used by our scheduler at the $i$th adaptation point.

By following the same principle, we can improve the execution demand bound $\gamma$. Let $H_i^e(j)$ denote the *execution demand history* of length $M$ and resolution $\psi$. $\psi$ is defined in terms of number of stream objects. The upper bound $\mathcal{E}_i$ on the number of processor cycles that can be requested by any sequence of $k$ consecutive stream objects after the $i$th adaptation point can be computed as follows

$$\mathcal{E}_i(k) = \inf_{0 \leq j \leq M} \{\gamma(j\psi + k) - H_i^e(j)\} \tag{12}$$

$$H_i^e(j) = e(l) - e(l - j\psi), \quad j = 0, 1, .., M \tag{13}$$

where $l$ is the total number of stream objects that have been completely processed up to the $i$th adaptation point. As a result we get the *dynamic execution demand bound*:

$$\hat{\gamma}_i(\delta) = \inf_{\delta \in [0, \hat{\alpha}_i(\tau)]} \{\mathcal{E}_i(\delta), \gamma(\delta)\} \tag{14}$$

$(\hat{\alpha}, \hat{\gamma})$ can used in all formulas of Subsection 3.3 in place of $(\alpha, \gamma)$.

## 3.5 A note on implementation

The DVS algorithm described in previous subsections can be implemented either in software or in hardware or using a combination of the two. For any implementation, a number of considerations have to be made. These include: (i) taking into account voltage/frequency transition overhead; (ii) working in discrete time; (iii) working with discrete frequency levels; (iv) determining the granularity and length of the workload history; (v) downloading the static bounds at run time if the application scenario changes, etc. Due to space constrains we skip these details here.

## 4. EXPERIMENTAL RESULTS

**Experimental setup:** To evaluate our DVS technique, we conducted several experiments using an MPEG-2 decoder application running in the setup shown in Figure 1. The simulator consisted of a SystemC transaction-level model in which $PE_1$ and $PE_2$ were modeled using the *sim-profile* configuration of the SimpleScalar ISS [1]. Table 1 lists a set of MPEG-2 video sequences using which we conducted our experiments. For each video, we collected traces corresponding to functions $x_i(t)$ and $e_i(k)$. By analyzing these traces on intervals of different lengths we obtained the curves $\alpha$ and $\gamma$ representing the whole set of videos. The resulting $\alpha$ is shown in Figure 3 for illustration. For estimating the energy consumption, we adopted the model from [5]: the energy $E \propto \int_0^{n\tau} v_{dd}^2 \cdot f\, dt = \sum_{i=1}^{n} (v_{dd,i})^2 f_i \cdot t \propto \sum_{i=1}^{n} (v_{dd,i})^2 f_i$, where $v_{dd,i}$ and $f_i$ are voltage and frequency values set by the scheduler at $i$th adaptation point, and $n$ is the total number of adaptation intervals. We assumed that $f_i \propto v_{dd,i}$.

**Qualitative examination:** Figure 4(a) shows a fragment of a frequency schedule produced by our DVS algorithm in an experiment and the resulting buffer fill levels. It illustrates how the two mechanisms of our method – the run-time monitoring of the buffer fill level and the dynamic worst-case bounds – work together to reduce the energy consumption. The dots on the frequency schedule plot show the adaptation points. By inspecting this plot we can make the following observations. Before time $t_1$, the load is low and the backlog is close to zero; hence, the processor runs at a low rate. At $t_1$ a burst starts arriving. In response to this burst the scheduler increases the clock rate, but not by too much: it lets the buffer fill up to some level and then tries to "balance" the clock rate at this level. However, shortly before $t_2$ the load abruptly increases even further and the buffer fills up very quickly (within one adaptation interval). Therefore, at $t_2$ to avoid a buffer overflow the scheduler increases the processor rate significantly, but only for a short time. In the interval from $t_2$ to $t_3$, the load is still high and the processor runs at a rate which approximately matches this load. At $t_3$ the buffer becomes almost full. Despite this fact, at $t_3$ our DVS algorithm decides to run the processor at a very low rate. How can this be possible? The answer to this is the effect of the dynamic bounds. At $t_3$ they tell the scheduler that the burst is over and it is safe to run the processor at a low rate because the next burst will not arrive very soon. Consequently, in the interval from $t_3$ to $t_4$, $PE_2$ runs at relatively low rates even though the buffer is nearly full during that interval. However, shortly before $t_4$, the scheduler again has to increase $PE_2$'s speed. This is because at $t_4$ the dynamic bounds tell the scheduler that a new burst might start arriving soon and therefore a sufficient space must be cleared in the buffer to accommodate this burst. Indeed, shortly after $t_4$ a new burst starts arriving and the adaptation cycle repeats.

**Quantitative comparison:** We compared our energy savings with those achieved by the DVS scheme published in Wu et al. [8]. Wu et al. uses a *PID controller* which tracks changes in the buffer fill level and correspondingly regulates processor's speed and voltage. This scheme is similar to ours in a sense that (i) it can handle both the stream burstiness and the data-dependent variability in the task execution demand; (ii) it is suitable for buffer-constrained architectures; and (iii) it also uses fixed adaptation intervals. Furthermore, to the best of our knowledge, the scheme of Wu et al. represents one of the advanced DVS techniques recently published. Therefore, we found it suitable for the comparison. From a user's perspective, the only difference between this scheme and ours is its *unpredictability* in terms of satisfying the specified QoS constraints. However, an implementation of this scheme might be associated with smaller SW/HW and energy overheads than of our DVS scheme.

We have implemented the PID controller as described in [8]. The adaptation interval lengths of the PID controller and that of our scheme were set to the same value, $\tau = 4.5$ms (which roughly corresponds to nine adaptations per video frame). In our scheme, the arrival history contained $N = 150$ samples with the resolution $\theta = \tau$, whereas the execution demand history contained $M = 200$ samples with the resolution $\psi = 1$. Figure 4(b) shows the results of this comparative study. In this figure, we refer to our scheme as *DVS* and to the PID controller scheme as *PID*. We simulated both schemes in two configurations. In one configuration, if the buffer was empty, $PE_2$ continued to run at the rate set at the latest adaptation point (i.e. some cycles were wasted in the idle state). In the other configuration, $PE_2$ was switched off completely for periods when there was nothing to process (i.e. no cycles were wasted). In Figure 4(b), the data corresponding to the latter configuration is indicated with dashed lines and with a suffix *_iDPM*. The switch-

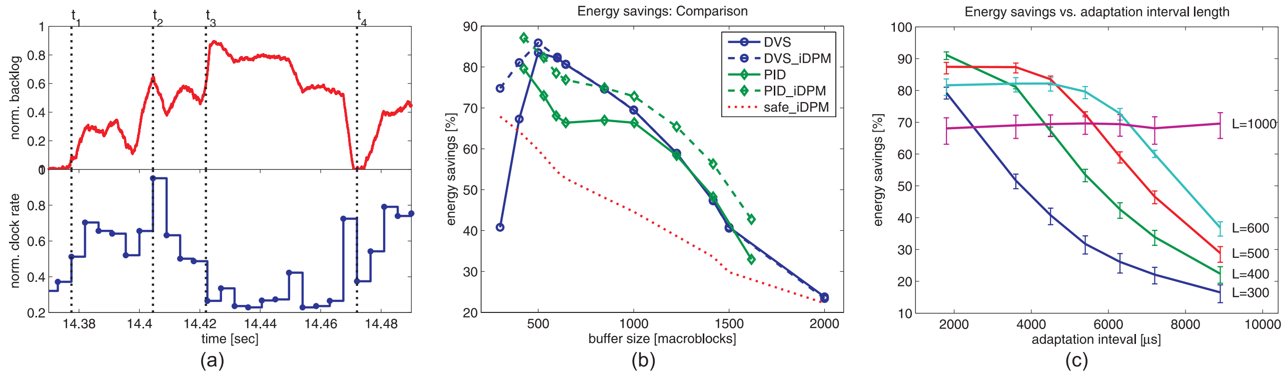

**Figure 4: Experimental results. In (b) our scheme is denoted as *DVS* and *DVS_iDPM*.**

ing on and off of $PE_2$ was assumed to occur in zero time under the control of an ideal ("oracle") DPM. Although unrealistic, such a configuration is useful for the analysis because it indicates how well a DVS technique can exploit the slack during low-load periods. As a baseline for measuring the energy savings we used the energy dissipated by $PE_2$ constantly running at the safe rate $f_{safe}$ computed for a given buffer size $L$ as described in Subsection 3.2. We also measured energy savings obtained by the "oracle" DPM with instantaneous on–off switching of $PE_2$ running at a fixed $f_{safe}$. The corresponding graph is shown in Figure 4(b) with a dotted line and is labelled as *safe_iDPM*.

By inspecting the plots in Figure 4(b) we can make the following observations. **(I)** For relatively small buffer sizes ($L \simeq 400 \ldots 1300$), both the DVS schemes (*DVS* and *PID*) result in more than 50% energy savings. With increasing buffer size the savings decrease. This is mainly because the energy consumption of the baseline architecture rapidly decreases when $L$ grows, since a lower $f_{safe}$ is needed to avoid buffer overflows. If $L$ is sufficiently large to completely average out the workload, *any* DVS scheme is likely to be not worthwhile. **(II)** In comparison to *safe_iDPM*, for $L \simeq 400 \ldots 1300$ the energy savings of both *DVS* and *PID* are 10…30%. This indicates that both techniques can effectively exploit the slack during low-load periods. We can also see that for $L > 600$ *DVS* fully exploits the slack (compared to *DVS_iDPM*), whereas *PID* potentially could perform better. **(III)** For $L > 450$ energy savings from *DVS* are at least as high as those from *PID*, while for $L < 450$, *PID* saves 15% more energy than *DVS*. This is the price for the hard QoS guarantees provided by *DVS*. If the user specifies $L$ as the buffer constraint, *DVS* guarantees that the maximum buffer fill level will never exceed $L$. In contrast, *PID* cannot guarantee this. In *PID* the user can only specify a *target* (not the maximum) buffer fill level. The PID controller will then try to keep the backlog at this level. Bursty workloads and stringent buffer constraints require the PID controller to quickly respond to workload changes. This leads to (large) oscillations around the target level during adaptations, which means that if the target level has been set improperly a buffer overflow may occur. As an example, the left most point in the *PID* graph was obtained by setting the target buffer fill level to 20 macroblocks, while the maximum backlog registered in the buffer for this setting was about 450 macroblocks. This problem might be less acute for smoother workloads and larger buffers.

**Energy savings vs. implementation overhead:** The estimated computational requirement of our DVS algorithm, for parameters $(\tau, N, \psi, M)$ set as described above, is about ∼0.5MIPS. This overhead scales linearly with the values of these parameters. It is also relatively low in comparison to the average workload imposed on $PE_2$ by a DVD-quality video stream (∼45MIPS in our case). Figure 4(c) shows the tradeoff plots between the adaptation interval length $\tau$ and the energy savings obtained by our scheme for different buffer sizes $L$. From Figure 4(c), we can see that smaller adaptation intervals lead to higher energy savings. To obtain an energy reduction with a small buffer size, that is comparable to what can be achieved with a larger buffer size, requires more frequent adaptations. This clearly results in higher run-time overhead, which is the price required for guaranteed QoS.

## 5. CONCLUDING REMARKS

In this paper we described a new DVS scheduling scheme specifically targeted towards processing multimedia streams on architectures with restricted buffer sizes. In contrast to previously proposed DVS schemes, our scheme provides hard QoS guarantees and accounts for both, the variability of the task execution demand and the burstiness of processed streams. Our experiments showed that the scheme achieves energy savings comparable to those obtained by previous approaches. The advantages of our scheme can be attributed to the novel combination of the offline worst-case workload characterization with the runtime improvement of the worst-case bounds. The implementation and runtime overhead of our scheme, although modest, might be slightly higher than that of previous schemes. However, this is the price that has to be paid for the predictability of the system, i.e. for its ability to provide QoS guarantees. Currently we are implementing a prototype version of the proposed DVS scheduler on an FPGA board for further evaluation of its performance in realistic application scenarios.

**Acknowledgements:** The Swiss Innovation Promotion Agency (KTI/CTI) supported the work of Alexander Maxiaguine through the project KTI 5500.2. Samarjit Chakraborty's work has been partially funded by the NUS URC grant R-252-000-190-112.

## 6. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer, 2001.

[3] V. Gutnik and A. P. Chandrakasan. Embedded power supply for low-power DSP. *IEEE Trans. on VLSI Syst.*, 5(4):425–435, 1997.

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

[5] Y.-H. Lu, L. Benini, and G. De Micheli. Dynamic frequency scaling with buffer insertion for mixed workloads. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11), November 2002.

[6] Z. Lu, J. Lach, M. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *ICCD*, 2003.

[7] T. Simunic, S. P. Boyd, and P. Glynn. Managing power consumption in networks on chips. *IEEE Trans. on VLSI Syst.*, 12(1):96–107, 2004.

[8] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS*, 2004.