Exploiting Inter and Intra Application Dynamism to Save Energy

Martijn Koedam, Sander Stuijk, and Henk Corporaal
Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands
{m.l.p.j.koedam, s.stuijk, h.corporaal}@tue.nl

Abstract. The dynamism inside applications can be exploited to save energy. A proactive scheduler that exploits this dynamism through Dynamic Frequency and Voltage Scaling (DVFS) has been presented in [1][2]. So far, the claimed energy savings of this scheduler have never been demonstrated on a real hardware platform. In this paper, we show for the first time that the proactive scheduler from [1][2] is able to realize the claimed energy savings. Our experimental results show that this scheduler reduces the energy consumption of a MP3 decoder running on a TI Omap3530 board by 18%. The proactive scheduler from [1][2] can only be used on a system that is running a single application. In this paper, we extend this scheduler such that it can deal with multiple applications that are running concurrently. Our scheduler exploits both inter and intra application dynamism to save energy while providing timing guarantees to all applications. Experimental results show that our scheduler is able to achieve the same energy savings, 38%, as an optimized version of the Linux ondemand scheduler when running two H.263 decoders concurrently. However, our scheduler achieves this result without any deadline misses; the ondemand scheduler fails 10% of its deadlines leading to a substantial quality loss.

***Index Terms*—Real-time scheduling, DVFS, embedded systems, energy reduction, system scenarios**

I. Introduction

Applications running on mobile embedded systems, e.g. smart phones, require powerful processors to deliver the Quality-of-Service which consumers expect from these systems. These processors however put a high strain on the battery life time of the system. The need to frequently recharge mobile systems immediately impacts the Quality-of-Experience of consumers. It is therefore important to develop techniques that allow mobile embedded systems to save energy, and thus increase the battery life time, while maintaining the desired Quality-of-Service (QoS). The resource requirements of many applications show large variations over time. In order to achieve the desired QoS, designers often allocate sufficient resources to be able to deliver the required QoS even under the worst-case situation. Exploiting the dynamic behavior of applications offers however great potential for saving resources and thus energy.

Several scheduling methods have been developed to exploit the dynamic behavior of applications. These can be divided in two classes: reactive methods and proactive methods. Reactive methods measure the past workload and use this information to adjust the processor frequency (e.g. using DVFS) to the lowest frequency with which the past workload could have been processed. However the past workload does not necessarily provide an accurate prediction of the upcoming workload. As a result, a reactive method may choose to run a processor at a too high or too low frequency. In the former case, the processor uses more energy then strictly required. In the latter case, the application misses its deadline and the QoS of the application is lowered. In contrast, proactive methods predict the upcoming workload based on information gathered both at design-time and run-time. This allows proactive methods to switch the processor to the lowest frequency with which the application is guaranteed to meet its timing constraints.

A proactive scheduling method that exploits the dynamic behavior of applications has been presented in [1][2]. So far, the claimed energy savings of this scheduler have never been demonstrated on a real hardware platform. In this paper, we show for the first time that this proactive scheduler is able to realize the claimed energy savings. Our experimental results show that this scheduler reduces the energy consumption of a H263 and MP3 decoder running on a realistic hardware platform which is used in several high-end smart phones. The proactive scheduler realizes for both applications a reduction in energy consumption of respectively 13% and 18% compared to a system that does not use DVFS (i.e. which is running at the lowest frequency needed to process the worst-case input without missing any deadlines). When compared to the state-of-the-art reactive scheduler used in Linux, the proactive scheduler has the same or lower energy consumption (H263 and MP3 respectively). The proactive scheduler achieves this result without any deadline misses (i.e. maintaining the QoS) while the reactive scheduler misses 2% of its deadlines.

The proactive scheduler from [2] can only be used on a system that is running a single application. In this paper, we extend this scheduler such that it can deal with multiple applications that are running concurrently. Our scheduler exploits both inter and intra application dynamism to save energy while providing timing guarantees to all applications. Experimental results show that our proactive scheduler is able to achieve the same energy consumption as an optimized version of the reactive scheduler used in Linux when running two H.263 decoders concurrently. Our scheduler achieves this result without any deadline misses whereas the ondemand scheduler misses 10% of its deadlines which leads to a substantial loss of quality.

The remainder of this paper is structured as follows. Sec. II presents related work. Sec. III describes our target platform. Sec. IV discusses the dynamic behavior of multimedia applications. Sec. V explains the approach used by our proactive scheduler to predict upcoming workloads. Sec. VI presents results for the proactive scheduler when scheduling a single ap-

plication on our platform. Sec. VII explains how this scheduler can be extended to deal with multiple concurrently running applications. Sec. VIII presents extensions to the platform and scheduler that would allow additional energy savings. Sec. IX concludes this paper.

II. Related Work

Many operating systems use a reactive scheduler in combination with DVFS to save energy. For example, the android OS uses a reactive scheduler called ondemand [3]. Reactive schedulers (e.g. [4], [3], [5], [6]) are easy to implement and result in reasonable energy savings without requiring any information from the applications that are scheduled using a reactive scheduler. However, reactive schedulers are not well suited for applications with real-time constraints (e.g. video or audio decoders). Reactive schedulers are often forced to run such applications at a frequency which is higher then strictly needed to meet the application timing constraints. This is due to the fact that the reactive scheduler cannot accurately predict the upcoming workload. It results however in a loss of energy which could potentially be avoided if the upcoming workload could be predicted in advance. Such a prediction of the upcoming workload is made by the proactive schedulers presented in [7], [1], [2], [5], [8]. Although these papers claim to achieve energy savings, these savings have never been demonstrated on a real hardware platform. In this paper, we show that the energy savings claimed in [1], [2] can be realized on a hardware platform that is used in high-end smart phones.

Many modern embedded systems are running multiple applications concurrently. Reactive schedulers such as the Linux ondemand scheduler [3] can be used in this context, but as explained before reactive schedulers may cause deadline misses or require more energy then strictly needed. Fixed priority schedulers, e.g. [9], [10], [11], are able to achieve a lower consumption compared to reactive schedulers while providing timing guarantees. These schedulers are typically only usable when all applications have periodic deadlines. Applications such as a H263 decoder do not fulfill this constraint. The multi-application schedulers presented in [11] and [12] can deal with a-periodic tasks, but these schedulers do not exploit the dynamic behavior of applications to save energy. As mentioned before, our experimental results indicate that substantial energy savings can be obtained when exploiting this dynamic behavior. Existing proactive schedulers (i.e. [7], [2], [5], [8]) can however only be used in systems that run a single application at the same time. Hence, these proactive schedulers cannot be used directly in a multi-application context. In this paper, we extend the proactive scheduler from [2] such that it can be used to concurrently schedule multiple time-constrained applications on a hardware platform.

III. Hardware Platform

The BeagleBoard [13] is used as our target platform. It is based on the TI Omap3530 System-on-Chip (SoC). This platform is used in many high-end embedded devices, like the Apple IPhone 3Gs and the Nokia N900. The Omap3530 SoC contains an ARM Cortex-A8 core, a TMS320C64x DSP Core, and a POWERVR SGX Graphics Accelerator. The BeagleBoard supports several DVFS operation points for the ARM core. These DVFS points are listed in Tab. I together with the average energy consumption of the complete board in each point. The energy consumption of the board is measured using a shunt resistor which is present in the power supply line. This allows us to measure the current drawn by the board. Using a custom build instrumentation amplifier and a data-logger connected to Matlab we measured the energy consumption of the board. In this paper, we focus on reducing the energy consumption of the ARM Cortex-A8 core. Therefore, we forced all other components in power down or sleep mode during our experiments. This avoids that components other then the ARM core influence our measurements. We validated our measurement setup using energy consumption data provided by Texas Instrument [14].

TABLE I
DVFS points and energy consumption BeagleBoard.

Frequency (MHz)	Voltage (V)	Consumption (mW)
125	0.975	$\sim$366
250	1.050	$\sim$456
500	1.200	$\sim$730
550	1.270	$\sim$785
600	1.350	$\sim$861

IV. Applications

A MP3 audio decoder and a H.263 video decoder are typical applications to be found in mobile multimedia systems such as the iPhone. This section discusses the dynamism that can be found in these applications. These applications will be used in the paper to demonstrate how this dynamism can be used to save energy.

A. MP3 Decoder

A MP3 bitstream can be broken down into a stream of audio frames. One frame represents 1152 output samples. At the start of each frame, a header indicates the parameters that need to be used when decoding this frame. Fig. 1 gives an overview of the steps in this decoding process. It starts with decompressing the bitstream using a Huffman decoder. The execution time needed by the Huffmann decoder depends strongly on the bitrate of the music. This bitrate is indicated in the header of each frame. The decoded data consists of two blocks called granules. Each granule holds 576 audio samples in the frequency domain. After decoding, the samples are dequantized and reordered. The execution time of both steps depends on the block type that is used. After reordering, the stream of samples is split into one or more channels. The number of channels depends on whether the encoded stream contains mono or stereo audio. Obviously, the number of channels has a large impact on the execution time of the MP3 decoder. The final step (IMDCT) in the MP3 decoder involves the transformation of the samples from the frequency domain to the time domain. This step shows no dynamic behavior.

B. H.263 Decoder

Fig. 2 shows the basic steps that are performed by a H.263 decoder. The decoding process of a H.263 decoder and a MP3 decoder are very similar. A H.263 decoder starts by decoding

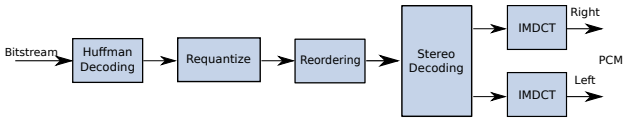

Fig. 1. Block diagram MP3 decoder.

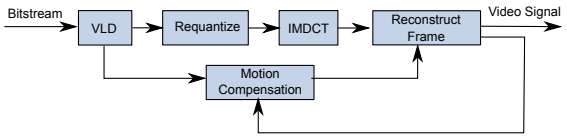

Fig. 2. Block diagram H.263 decoder.

the bitstream using a variable-length decoder. Inside the header of the bitstream, several parameters such as the number of macro-blocks inside a frame, the picture encoding type, and the frame rate are present. These parameters have a large influence on the execution time of the whole decoder. Since these parameters are part of the header of each frame, their value will be known early on in the frame decoding process. As explained in the next section, we can use the values of these parameters to predict the execution time of the application. This information can then be used to adjust the voltage and frequency of the processor which results in a lower energy usage.

V. System Scenarios

Applications that are running on embedded multimedia systems are full of dynamism. Their execution time depends, amongst others, on the input data. When such an application is running on a given platform it may encounter different run-time situations. A run-time situation is a piece of system execution that is treated as an atomic unit. Each run-time situation has an associated cost (e.g. resource usage). The execution of the system is a sequence of run-time situations. The current run-time situation is only known at the moment that it occurs. However, a set of so called scenario parameters (i.e. variables in the application) can be used to predict the next run-time situation. The knowledge that this run-time situation will occur in the near future can be used to adapt the system settings (e.g. processor frequency) to this run-time situation. The number of distinguishable run-time situations from a system is exponential in the number of scenario parameters. To avoid that all these situations must be handled separately at run-time, several run-time situations should be clustered into a single system scenario. This clustering presents a trade-off between the optimization quality and the run-time overhead of the scenario exploitation.

A general methodology to identify and exploit scenarios has been described in [2]. This methodology consists of a combination of design-time and run-time steps. At design-time, the scenario methodology starts with identifying the potential scenario parameters and clustering the run-time situations into scenarios. During this clustering, a trade-off must be made between the cost of having more scenarios (e.g. larger code size, more scenario switches) and the cost of grouping run-time situations in one scenario (e.g. over-dimensioning of the system for certain run-time situations). The second design-time step involves the construction of a predictor that at run-time observes the scenario parameters and that uses their observed values to select a scenario at run-time. Finally, at design-time an algorithm is constructed that is used at run-time to decide when to switch between scenarios. This algorithm does not only decide when it is required to switch between different scenarios. When switching, it also changes the system settings of the old scenario to the settings required for the new scenario (e.g. it may change the processor frequency).

VI. Single application

In [2], Gheorghita presented an implementation of the scenario methodology and associated proactive scheduler. He demonstrated the energy savings of this methodology using a platform simulator. So far, these savings have never been demonstrated on a real platform. Results for such a validation will be presented in this section.

In order to test the proactive scheduler of Gheorghita on our platform, we had to make one important change to his implementation. The scenario method and implementation described in [2] assume a platform with many DVFS points (i.e. one point per MHz). In practice, many platform have only a limited set of DVFS points. Our platform has in fact only 5 DVFS points over its entire range from 125-600 MHz (see Tab. I). The assumption made by the scenario method from [2] may cause the generation of a non-optimal set of scenarios for our platform. The method may generate and predict scenarios that cannot be exploited on the target hardware. Consider as an example a situation in which the method forms different scenarios running at 300, 400 and 500 MHz frequency while the actual hardware is only able to run at 500 MHz. The generated scenario detector will need to distinguish these three scenarios although they effectively all need to run on the same frequency. As a result, the detector will be more complex then strictly needed. We modified the implementation from [2] to take the limited number of DVFS points in our platform into account. Due to the limited number of DVFS points in our platform, we find a smaller set of scenarios compared to [2].

To validate the scenario methodology from [2], we performed experiments with a MP3 decoder and a H263 decoder. In both experiments, we compare the proactive scheduler proposed by [2] to the Linux ondemand scheduler [3] which is is a state-of-the-art reactive scheduler. We also compare the proactive scheduler to a worst-case scheduler which does not use DVFS. Instead, it keeps the frequency fixed at the lowest frequency needed to meet the timing constraints of the application under its worst-case behavior.

MP3 decoder. Gheorghita validated his method using an MP3 decoder. We used the same source code [15] as used by Gheorghita to perform our experiments. The scenario methodology discovers three different scenarios in this application. These scenarios capture the dynamism inside the various blocks of the MP3 decoder (see Sec. IV) and they are exploited by the proactive scheduler to save energy. Tab. II shows the average power consumption of the proactive scheduler and our two reference schedulers. The third column shows the resulting energy usage relative to the worst-case scheduler. The results

TABLE II
ENERGY CONSUMPTION MP3 DECODER.

Scheduler	Avg power (mW)	Energy (%)	Deadline Misses
Worst-case	766	100%	0%
Reactive	632	83%	2%
Proactive	627	82%	0%

TABLE III
GRANULARITY REACTIVE SCHEDULER - MP3 DECODER.

Granularity	Avg power (mW)	Energy (%)	Deadline Misses
100ms	647	85%	1%
300ms	632	83%	2%
900ms	658	86%	6%

TABLE IV
ENERGY CONSUMPTION H263 DECODER.

Scheduler	Avg power (mW)	Energy (%)	Deadline misses
Worst-case	463	100%	0%
Reactive	404	87%	9%
Proactive	405	87%	0%

show that our proactive scheduler is able to achieve an 18% energy saving compared to the worst-case scheduler. In [2], an energy saving of 12% has been reported for the proactive scheduler. This result is in-line with the saving found by our measurements on the BeagleBoard. From this we conclude that the claim made in [2] is valid and that it is possible to achieve actual energy savings using a proactive scheduler. Tab. II also shows that the relative energy consumption of the reactive scheduler is 1% higher than our proactive scheduler (i.e. the energy saving obtained with the reactive scheduler is smaller compared to the proactive scheduler). Despite this higher energy consumption, the reactive scheduler still misses 2% of its deadlines while the proactive scheduler does not fail any deadline. This shows that the proactive scheduler is able to achieve energy savings without loss of quality.

The number of deadline misses of the reactive scheduler could potentially be reduced by changing the granularity with which the scheduler is invoked. By default, the scheduler is invoked once every 300 ms. Tab. III shows the effect when changing this granularity to 100 ms and 900 ms. The results shows that a smaller granularity (100 ms) leads to fewer deadline misses, but a larger energy consumption. This is to be expected as the overhead of the scheduler, which costs energy, becomes larger. The results also show that a larger granularity (900 ms) results in more deadline misses and an increased energy consumption. This is caused by the fact that the scheduler is not invoked frequently enough. Whenever it is invoked, there is a reasonable chance that a deadline has been missed. In this situation, the scheduler will switch to a very high frequency in order to avoid deadline misses in the future. This will however cause a large increase in the energy consumption. On the next invocation, the scheduler will see that the workload does not justify the chosen frequency and it will lower the frequency often below the frequency needed to meet all deadlines. At this moment, the whole process repeats itself. It is this constant switching between too high and too low frequencies that causes the scheduler to increase its energy consumption compared to a reactive scheduler which is invoked every 300 ms. The results in Tab. III show that a granularity of 300 ms for the reactive scheduler provides the lowest energy consumption. We already saw earlier in Tab. II that even in this situation, the proactive scheduler has a lower energy consumption and better quality. This confirms again that the proactive scheduler is able to exploit the inter application dynamism to save energy.

H263 decoder. As a second experiment, we measured the energy consumption of a H263 decoder running on our hardware platform. Tab. IV shows the power consumption and relative energy consumption when using the proactive scheduler from [2] as well as our two reference schedulers. The results of this experiment are similar to the results from the MP3 decoder experiment. Both the proactive and reactive scheduler are able to achieve an energy saving of 13% compared to the worst-case scheduler. The reactive scheduler achieves this result with some quality loss (i.e. it misses 10% of its deadlines). The proactive scheduler can achieve this saving without any quality loss. Similar to the MP3 decoder experiment, we adjusted the granularity with which the reactive scheduler could be invoked. These results showed that a granularity of 300 ms provides the lowest energy consumption. Decreasing the granularity would lead to a higher quality, but also a higher energy consumption. From this experiment, we can conclude that the proactive scheduler outperforms the reactive scheduler as it achieves the same energy reduction, but without any quality loss.

VII. MULTIPLE APPLICATIONS

The proactive scheduler presented in [2] can only be used when running a single application on the platform. Modern embedded platforms typically run multiple applications concurrently. In this section, we explain how the proactive scheduler from [2] can be extend to handle multiple, time-constrained applications. Similar to the single application proactive scheduler discussed in the previous section, this scheduler uses the concept of system scenarios to capture the dynamic behavior of applications and to save energy.

A. Problem statement

We start by formalizing our multiple application scheduling problem. Assume a set $A = \{A_1, A_2, \cdots, A_n\}$ of n running applications. Each application A_i has a worst-case workload Lwc_i. Whenever an application A_i is running, a run-time situation is active in a particular scenario. This run-time situation has a deadline D_i, and remaining workload W_i (which is scenario dependent). The start time of a run-time situation is the deadline of the previous run-time situation. The workload of a run-time situation (i.e. W_i) and the next deadline (i.e. D_i) become available at the start of the run-time situation. An active application can be preempted at any time. This causes a context switch overhead CW_{oh} (in cycles). Whenever the scheduler is invoked, it may decide to switch the frequency of the processor which causes a speed transition overhead SW_{oh}. Both overheads needs to be taken into account by the scheduler. The scheduler is invoked whenever the running application finishes its work for its current run-time situation

or when the deadline of another application is reached. At that moment, the scheduler can choose a new DVFS operating point. This DVFS operating point should be chosen in such a way that all applications can meet their deadline while also minimizing the overall energy consumption of the platform.

B. Multiple Application Scheduler

Whenever our multiple application scheduler is invoked, it must take the following two decision: (1) it must select the next application to be executed (activated), (2) it must select the DVFS operating point (i.e. processor frequency). To take the first decision, most uni-processor platforms use an Earliest Deadline First (EDF) [16] or Rate Monotonic (RM) [17] scheduler. We choose to use an EDF scheduler as it supports a high utilization bound (i.e. close to 1.0) which is important from an energy point of view. An RM scheduler does not allow a utilization bound above 0.69 [18]. Hence, an RM scheduler requires an earlier switch to a higher frequency compared to an EDF scheduler. In other words, an EDF scheduler is preferred over an RM scheduler because it allows larger energy savings.

Our proactive multiple application scheduler start with invoking the EDF scheduler. This scheduler selects the application A_i from the set of applications A that has the earliest deadline D_i and which still has work left before this deadline (i.e. $W_i > 0$). The EDF scheduler also determines the deadline D_{EDF} at which the next invocation of our proactive scheduler needs to take place. This deadline is equal to the minimum of the next deadline of the selected application (i.e. A_i) and the deadlines of all other applications. Once this deadline is computed, the EDF scheduler ends and our proactive scheduler invokes the DVFS Selection and Switching (DVFS^3) algorithm. This algorithm is responsible for selecting the DVFS operating point and performing a switch between two DVFS points whenever needed. The DVFS^3 algorithm has to take into account the workload that is currently available, the workload which may be offered in the future, and the time that is available to perform all this work. As explained in Sec. VII-A, our proactive scheduler and thus the DVFS^3 algorithm must take the switching overhead into account as a switch between two DVFS points can take several milliseconds [19][20].

The pseudo-code of our DVFS^3 algorithm is shown in Algorithm 1. We will explain the working of this algorithm using the example situation shown in Fig. 3. In this example, the DVFS^3 algorithm is activated at time $S_{activate}$. At this moment, the applications A_1, A_2, and A_3 all need to perform some work before their respective deadlines. Application A_2 has the earliest deadline (i.e. $D_{EDF} = D_2$). The EDF scheduler, which was invoked just before the DVFS^3 algorithm, has therefore decided that this application will be activated next. The DVFS^3 algorithm should now determine the DVFS operating point (i.e. processor frequency) that needs to be used. The algorithm starts by computing the total workload, *work*, that is available at $S_{activate}$. For our running example (see Fig. 3), this workload is equal to the total area of the dashed boxes. When computing *work*, the algorithm also determines the last deadline, $T_{last\ deadline}$, of all running applications (red line in Fig. 3). Looking beyond this deadline does not give any useful information as it is unknown at this moment (i.e. at $S_{activate}$) which scenarios the applications will be execution after $T_{last\ deadline}$. Next, the algorithm computes in lines 11-13 an upper bound on the workload that needs to be performed before the last deadline (i.e. before $T_{last\ deadline}$). Before $T_{last\ deadline}$, some applications may have already started a next run-time situation. At the $S_{activate}$, the DVFS^3 algorithm does however not know the scenario of these run-time situations. Hence, it does not know the scenario-predicted worst-case workload of these run-time situations. Therefore the algorithm must assume that the workload of these run-time situations is equal to the worst-case scenario. In our example (see Fig. 3), the total workload before the last deadline is equal to the sum of the dashed and dotted boxes. The algorithm continues in line 14 by computing the minimal frequency needed to perform all available work (*work*) before the first deadline. The computation takes the context switching overhead (CW_{oh}) and the DVFS switching overhead (SW_{oh}) into account. This computation uses the Nearest DVFS Point ($NDP()$) function. This function returns the frequency of the nearest DVFS points that has a frequency equal or higher as the supplied frequency. Line 15 of our algorithm performs the same computation, but without the DVFS switching overhead. The frequencies computed in line 14 and 15 assume that all available work needs to be finished before D_{EDF}. It might sometimes be better to postpone some of this work till after this deadline as this may allow the platform to run at a lower frequency. Whether this is possible depends not only on the available work, but

Algorithm 1 DVFS Selection and Switching (DVFS^3)

1: $work \leftarrow 0$
2: $workld \leftarrow 0$
3: $T_{last\ deadline} \leftarrow 0$
4: **for all** $A_i \in A$ **do**
5: $\quad work \leftarrow work + W_i$
6: $\quad$ **if** $D_i > T_{last\ deadline}$ **then**
7: $\quad\quad T_{last\ deadline} \leftarrow D_i$
8: $\quad$ **end if**
9: **end for**
10: $workld \leftarrow work$
11: **for all** $A_i \in A$ **do**
12: $\quad workld \leftarrow workld + Lwc_i \cdot (T_{last\ deadline} - D_i)$
13: **end for**
14: $f_{first\ with\ oh} \leftarrow NDP(\frac{work+SW_{oh}+CW_{oh}}{D_{EDF}-S_{activate}})$
15: $f_{first\ w/o\ oh} \leftarrow NDP\left(\frac{work+CW_{oh}}{D_{EDF}-S_{activate}}\right)$
16: $f_{last\ with\ oh} \leftarrow NDP(\frac{workld+SW_{oh}+CW_{oh}}{last\ deadline-S_{activate}})$
17: $f_{last\ w/o\ oh} \leftarrow NDP\left(\frac{workld+CW_{oh}}{last\ deadline-S_{activate}}\right)$
18: $f_{min\ w/o\ oh} \leftarrow min\left(f_{first\ w/o\ oh}, f_{last\ w/o\ oh}\right)$
19: **if** $f_{min\ w/o\ oh} \neq f_{current}$ **then**
20: $\quad f_{min\ with\ oh} \leftarrow min\left(f_{first\ withoh}, f_{last\ with\ oh}\right)$
21: $\quad$ **if** $\frac{(SW_{oh}+CW_{oh})}{min\left(f_{min\ w/o\ oh}, f_{min\ with\ oh}\right)} < (D_{EDF} - S_{activate})$ **then**
22: $\quad\quad f_{current} \leftarrow f_{oh}$
23: $\quad$ **end if**
24: **end if**

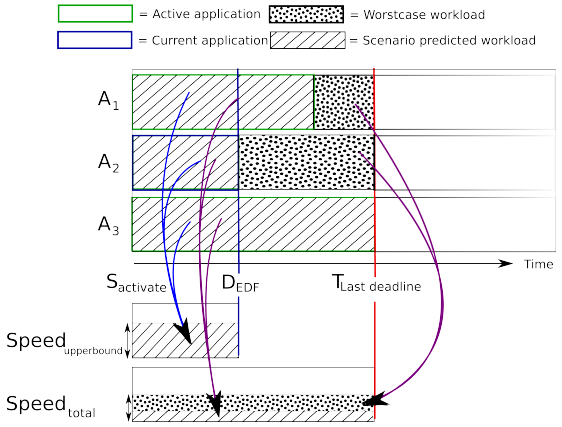

Fig. 3. DVFS^3 algorithm.

also on the additional work that may arrive before the last deadline. In lines 11-13, the algorithm already computed an upper bound on the workload that may have to be processed before this last deadline. Lines 16 and 17 use this upper bound (*workld*) to compute the frequency needed to finish all work before the last deadline. Similar to the earlier computation, two frequencies are computed. One for the situation in which no DVFS switch needs to be performed ($f_{last\ w/o\ oh}$) and one which considers the DVFS switching overhead ($f_{last\ with\ oh}$). Obviously, the processor only needs to run at the minimum of the four frequencies computed so far as this will always allow all applications to meet their timing constraint. Two of these frequencies require a DVFS switch and the other two require no switch. Lines 18-24 determine whether a DVFS switch is needed. If the required frequency ($f_{min\ w/o\ oh}$) matches the current frequency $f_{current}$, no switch (and therefore no overhead) is required. This avoids that the processor goes to a higher frequency only for the overhead. As a last check, the algorithm checks in line 21 whether the time available till the next deadline is larger then the time needed to switch the frequency. It is obviously not needed to make a switch if a switch would result in a deadline miss.

C. Experimental Evaluation

Similar as in the single application experiments (see Sec. VI), we compare our proactive scheduler to the Linux on-demand (reactive) scheduler and a worst-case scheduler. The granularity with which the reactive scheduler is invoked is experimentally set to the value needed to get the largest energy savings. (Note that the default settings of the ondemand scheduler may provide worse results.) Similar to our proactive scheduler, the reactive scheduler must be combined with an algorithm that decides which application is activated. Due to a constraint in the Linux kernel, it is not possible to use the EDF algorithm for this purpose. Therefore, we combined the ondemand scheduler with the Completely Fair Scheduler (CFS). This schedulers tries to give an equal amount of processing time to all running applications. Since this scheduler does not consider the deadline of an application, it may cause more deadline misses then an EDF algorithm. Since the ondemand scheduler chooses the highest DVFS operating point when a deadline miss occurs, the combination of ondemand and CFS may show a higher energy consumption then a combination of ondemand and EDF. Since the latter combination does not work due to limitation in the Linux kernel, we choose to anyhow use the combination of ondemand and CFS for our reactive scheduler. This gives us the best realizable comparison between a proactive and reactive scheduler. We also compare our proactive scheduler to a worst-case scheduler which does not use DVFS. This scheduler keeps the frequency fixed at the lowest frequency needed to meet the timing constraints of the application under its worst-case behavior and it uses the EDF algorithm to determine which application is activated. As a third comparison, we created a Constraint Programming algorithm that computes a lower bound on the energy consumption of all possible schedules that meet the timing constraints of all applications. Note that the problem of finding the optimal schedule is NP-hard [12]. It is therefore not possible to find the optimal solution in a reasonable amount of time. Several assumptions are made in our algorithm to limit its run-time. It is assumed that both the transition delay and the context switch overhead are zero. These approximation can only result in an underestimation of the energy usage. Hence, the energy consumption computed by our algorithm is a valid lower bound.

As explained in Sec. VII-B, our proactive multiple application scheduler consists of a combination of an EDF scheduler and our DVFS^3 algorithm. This scheduler is implemented in a small program that runs on our hardware platform. Using the real-time scheduler from the Linux kernel, the program is given a time slice that is longer then the duration of any experiment. This allows the program to run without interruptions. Hence, only our scheduler will be invoked during the experiments. The Linux scheduler will not be activated. The program schedules and executes the different applications as threads spawned by our scheduler.

Two H263 decoders. We measured the energy usage of our platform when running two H263 decoders concurrently. Fig. 4 show the energy usage of the various schedulers relative to energy usage of the worst-case scheduler. The figure shows that the proactive and reactive scheduler achieves similar energy savings (around 38%). The proactive scheduler realizes these savings without any quality loss. The reactive scheduler misses 10% of its deadlines. Hence, its quality is considerably degraded compared to our proactive scheduler. This shows that our proactive scheduler is able to achieve the same savings as a state-of-the-art reactive scheduler, but while delivering a much higher quality.

Fig. 4 also shows that the proactive scheduler gets within 6% of the lower bound. This shows that our proactive scheduler and in fact also the reactive scheduler enforce a schedule which is close to the schedule with minimal energy usage. Hence, the proactive scheduler effectively exploits the inter and intra application dynamism to save energy.

Scaled H263 and MP3 decoders. To mimic a real multimedia system, we tried to run a combination of one H263 decoder and two MP3 decoders on our platform. Unfortunately, our platform does not have enough performance to run these three applications together. This forced us to scale the workload of

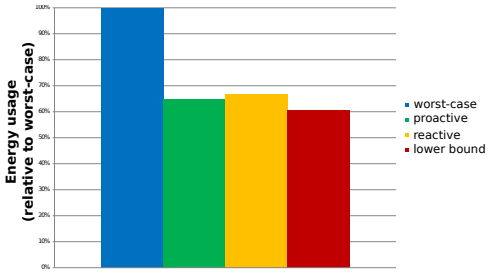

Fig. 4. Two H263 decoders.

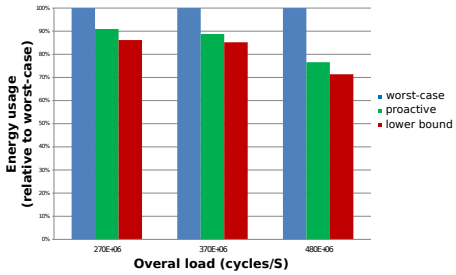

Fig. 5. Scaled workload - one H263 decoders and two MP3 decoder.

this set of applications to a range which could be ran on the platform. Since we scale all application workloads in the same way, this scaling will not affect the relative performance of the various schedulers. Therefore we felt that it is justified to perform this scaling. In total, we created three sets of our scaled applications. The application in these sets have a combined workload of respectively $270 \cdot 10^6$, $370 \cdot 10^6$ and $480 \cdot 10^6$ cycles/second.

Fig. 5 shows the energy usage of the various workloads for the different schedulers. Note that in this comparison we excluded the reactive scheduler. This is due to a problem in the Linux kernel which made it impossible to schedule our scaled workload using the ondemand scheduler. It can be seen in Fig. 5 that the schedulers achieve different energy savings when the workload changes. This is due to the limited set of DVFS points available in our platform. For some workloads, there are more options to select a favorable DVFS operating point compared to some other workloads. In our experiment, we achieve a maximal saving of 25% when the dynamism between the applications is maximal (i.e. in the workload $270 \cdot 10^6$ cycles/s). This shows that when the space between the average load and the worst-case load is larger, a larger energy saving can be achieved. The results also show that even when we are are very close to the worst-case point, 480 MHz average versus 550 MHz worst-case, we can still safe around 10% on energy. Finally, the results show that our scheduler is always within 3-6% from the lower bound. This experiment reconfirms our earlier conclusion that the proactive scheduler can effectively exploit the inter and intra application dynamic behavior.

VIII. Extensions to HW Platform and Scheduler

The experimental results presented in the previous sections are all measured on the BeagleBoard (see Sec. III). The exploited scenarios have been discovered using the scenario methodology from [2]. In this section, we present several changes that can be made to the hardware platform and the scenario methodology in order to improve the energy savings of the system. Since these changes involve changes in the hardware, we were not able to validate the impact of these changes on the real hardware. The experimental results presented in this section are therefore obtained using an energy simulator which we developed for this platform. Before running the experiments outlined in this section, we validated the simulator using the data obtained from the experiments presented in the previous sections.

More DVFS points. The BeagleBoard has five DVFS points (see Tab. I). In order to provide timing guarantees, the proactive scheduler always chooses the smallest frequency that is large enough to process the upcoming workload. Increasing the number of DVFS points allows the scheduler to use a frequency that is closer to the actual frequency needed to process this workload. However, an increase in the number of DVFS points could lead to an increase in the number of DVFS switches. As a result, the transition delay when switching between DVFS point could negatively impact the energy savings. To analyze this effect, we increased the number of DVFS points in our simulator as compared to the actual BeagleBoard. Furthermore, we distributed these DVFS points evenly over the entire frequency range (125-600 MHz). Line 1 in Fig. 6 shows the energy usage of three H263 decoder running concurrently on the platform when varying the number of DVFS points. The time needed to switch between two DVFS points, i.e. the transition delay, is kept the same as on the real BeagleBoard (i.e. 300000 cycles). The figure shows that a board with 7 DVFS points would result in the highest energy savings. The figure also shows that the switching overhead impacts the savings when more then 7 DVFS points are used. Comparing line 1 in Fig. 6 to the 5 DVFS points used in the real board (see left-hand side of Fig. 6), it is clear that equally distributing these 5 DVFS points is more energy efficient then the original distribution. This result can be expected considering the fact that the real board has only 3 DVFS points in the lowest 83% of its frequency range. Increasing the number of DVFS points within this part of the range provides more opportunities to select a suitable frequency and hence run the board at a lower frequency.

Decreasing the transition delay. Line 1 in Fig. 6 shows that the transition delay becomes significant when more then 7 DVFS points are used. The transition delay as used by the Linux kernel could theoretically be lowered. According to the TI datasheets, it can be lowered to 30000 cycles. Line 4 in Fig. 6 shows the energy usage of the two H263 decoders when varying the number of DVFS points. Compared to line 1, it is clear that a lower transition delay allows for additional energy savings of almost 7%. Moreover, decreasing the transition delay makes it also possible to use more DVFS points to save additional energy.

Scenario-based worst-case workload. The DVFS^3 algorithm uses the worst-case workload of an application when estimating the total workload till the last known deadline. This worst-case workload can be very pessimistic. Consider as an example a video decoder that can process videos with

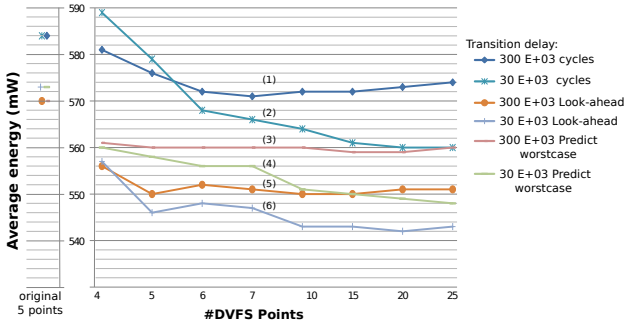

Fig. 6. Different DVFS points and transition delays.

resolutions up-to full-HD. The worst-case workload will be based on a full-HD video. This worst-case load will however never be reached when processing a video with a lower resolution. Some properties of the application are constant throughout the run-time of the application (e.g. the resolution of a video). This information could be used to provide a tighter bound on the worst-case workload. This can be seen as adding an additional level of scenarios of which one scenario is selected at the start of the application. Line 3 in Fig. 6 shows the energy usage of our setup when our DVFS^3 algorithm uses a resolution-specific worst-case workload. The figure shows that a scenario-based worst-case workload allows an additional energy saving of 2% compared to the original setup on the BeagleBoard.

Advance scenario prediction. The scenario methodology from [2] only predicts the scenario of the current run-time situation. Predicting in advance the scenarios of future run-time situations would allow the DVFS^3 algorithm to use tighter estimates on the worst-case workload of upcoming run-time situations. Line 5 and 6 in Fig. 6 show the energy usage of our setup when predicting the scenario of the next run-time situation. The result shows that predicting scenarios in advance leads to a 7% energy saving as compared to the original setup. This shows the potential of predicting scenarios in advances. Extending the scenario framework to make such a prediction is however left as future work.

IX. Conclusion

In this paper we validated the proactive method from [1][2] on a real hardware platform. Our results show that this method performs better then an existing state-of-the-art reactive method. The proactive method from [1][2] only works on single applications. This paper proposes a proactive multi-application scheduler. It schedules multiple real-time applications concurrently while minimizing the energy consumption of the system. Experimental results show that our proactive scheduler is able to achieve the same energy consumption as an optimized version of the Linux ondemand scheduler when running two H.263 decoders concurrently. Both schedulers realize a 38% reduction in energy consumption compared to a worst-case scheduler. Our scheduler achieves this result without any deadline misses whereas the ondemand scheduler misses 10% of its deadlines which leads to a substantial quality loss. This shows that exploiting inter and intra application dynamism allows for significant energy saving without loss of real-time performance. As future work, we consider the development of a hybrid scheduler that allows DVFS energy saving on a system with mixed real-time and non-real-time applications.

References

[1] S. Gheorghita, T. Basten, and H. Corporaal, "Application scenarios in streaming-oriented embedded-system design," *IEEE Design and Test of Computers*, vol. 25, no. 6, pp. 581–589, 2008.

[2] S. V. Gheorghita, "Dealing with dynamism in embedded system design: Application scenarios," Ph.D. dissertation, Eindhoven Univeristy of Technology, 2007.

[3] V. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proceedings of Linux Symposium*, 2006, pp. 223–238.

[4] K. Govil, E. C., and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power CPU," in *Proceedings of the International Conference on Mobile computing and networking, MobiCom '95*. New York, NY, USA: ACM, 1995, pp. 13–25.

[5] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED '98*. New York, NY, USA: ACM, 1998, pp. 76–81.

[6] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proceedings of the Conference on Operating Systems Design and Implementation, OSDI '94*. Berkeley, CA, USA: USENIX, 1994, pp. 13–23.

[7] E. Y. Chung, L. Benini, and G. De Micheli, "Contents provider-assisted dynamic voltage scaling for low energy multimedia applications," in *Proceedings of the International Symposium on Low Power Electronics and Design, 2002. ISLPED '02*. New York, NY, USA: ACM, 2002, pp. 42–47.

[8] R. Sasanka, C. J. Hughes, and S. V. Adve, "Joint local and global hardware adaptations for energy," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '02*. New York, NY, USA: ACM, 2002, pp. 144–155.

[9] F. Gruian, "Hard real-time scheduling for low-energy using stochastic data and dvs processors," in *Proceedings of the International Symposium on Low Power Electronics and Design, ISLED '01*. New York, NY, USA: ACM, 2001, pp. 46–51.

[10] W. Kim, J. Kim, and S. L. Min, "Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis," in *Proceedings of the International Symposium on Low Power Electronics and Design, ISLED '03*. New York, NY, USA: ACM, 2003, pp. 396–401.

[11] B. Mochocki, X. Hu, and G. Quan, "Practical on-line dvs scheduling for fixed-priority real-time systems," in *Proceedings of the International Symposium on Real Time and Embedded Technology and Applications, RTAS '05*. Los Alimatos, CA, USA: IEEE, 2005, pp. 224–233.

[12] S. Zhang, K. S. Chatha, and G. Konjevod, "Approximation algorithms for power minimization of earliest deadline first and rate monotonic schedules," in *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED '07*. New York, NY, USA: ACM, 2007, pp. 225–230.

[13] Beagle Board, http://www.beagleboard.org/.

[14] Texas Instrument, "Power estimation spreadsheet," 2010, http://processors. wiki.ti.com/index.php/ OMAP3530_ Power_Estimation_Spreadsheet.

[15] K. Lagerstrom, "Design and implementation of an MPEG-1 layer III audio decoder," Master's thesis, Chalmers University of Technology, Sweden, 2001.

[16] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[17] G. C. Buttazzo, "Rate monotonic vs. EDF: Judgment day," *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.

[18] G. Buttazzo, Ed., *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*, ser. Real-Time Systems. Santa Clara, CA, USA: Springer-Verlag, 2004.

[19] "Linux 2.6.32 kernel source."

[20] "Omap3530 specifications," http://www.ti.com/lit/gpn/omap3530.