An Efficient Frequency Scaling Approach for Energy-Aware Embedded Real-Time Systems

Christian Poellabauer[1], Tao Zhang[2], Santosh Pande[2], Karsten Schwan[2]

[1] Computer Science and Engineering, University of Notre Dame
cpoellab@cse.nd.edu
[2] College of Computing, Georgia Institute of Technology
{zhangtao, santosh, schwan}@cc.gatech.edu

Abstract. The management of energy consumption in battery-operated embedded and pervasive systems is increasingly important in order to extend battery lifetime or to increase the number of applications that can use the system's resources. Dynamic voltage and frequency scaling (DVFS) has been introduced to trade off system performance with energy consumption. For real-time applications, systems supporting DVFS have to balance the achieved energy savings with the deadline constraints of applications. Previous work has used periodic evaluation of an application's progress (e.g., with periodic *checkpoints* inserted into application code at compile time) to decide *if* and *how much* to adjust the frequency or voltage. Our approach builds on this prior work and addresses the overheads associated with these solutions by replacing periodic checkpoints with iterative checkpoint computations based on predicted best-, average-, and worst-case execution times of real-time applications (e.g., obtained through compile-time analysis or profiling).

1 Introduction

Motivation. Energy management has become a central issue in the embedded systems domain, where an increasing number of devices, including personal digital assistants, cell phones, medical equipment, and solar-powered systems, are supported by rechargeable batteries. If applications have stringent requirements for high performance or real-time guarantees, the energy consumption of these devices has to be carefully balanced with the resource utilization and application needs. Efficient energy management can result in reduced battery specifications (resulting in smaller and lighter devices), maximized battery lifetime, and increased mission duration. Fortunately, embedded applications can take advantage from a multitude of novel energy saving techniques. At the hardware level, consider the StrongARM SA11xx processors, the Intel XScale 80200, or the Transmeta Crusoe with LongRun, all of which support the run-time selection of different frequency or voltage levels [10, 14]. At the network level, wireless cards and disks are built with support for multiple power modes, i.e., these devices can be switched into a power-saving mode when idle [6]. Finally, at the application level, energy-aware transcoding and adaptation techniques [12, 17] reduce the

computation or communication needs, and therefore, the energy requirements of these applications. The energy management approach addressed in this paper is the frequency and voltage scaling capabilities of modern mobile processors. Consider a multimedia application in which a mobile device receives one or more video and audio streams that have to be replayed with certain requirements for constant rates and maximum jitter to ensure sufficient quality. This requires that the device allocates sufficient processor and network resources to these applications. However, especially with wireless communications, it is likely that video and audio frames will arrive in bursts, where the receiving device will buffer incoming data until their replay time has arrived. Based on the desired replay rate, a deadline for the replay of each frame can be derived. If the CPU is not fully utilized, frequency or voltage scaling can be used to slow down the execution of the video and audio players, therefore reducing the energy consumption of the device, while still ensuring the timely replay of video and audio.

Problem statement. Previous work has introduced approaches to dynamically change the speed or voltage at different layers of an embedded system, e.g., as compile-time tool or as operating system extension. These approaches predict application run-time – e.g., from information collected through code analysis – and compute a clock frequency or voltage accordingly. However, variations in the run-time, caused by changes in the application behavior, input variables, or by resource scarcity, can lead to mispredictions, resulting in missed deadlines or inefficient energy management. Therefore, these approaches monitor the progress of a real-time application, e.g., by inserting *checkpoints* [3] or *hints* [1] into the application code or by comparing the progress to statistical application behavior [5]. As a result, speed or voltage are adjusted to compensate for these variations. Our approach builds on this prior work and addresses the overheads associated with these solutions, which stem from two sources: (a) cost of checkpointing and progress evaluation and (b) cost of frequency and voltage adjustments. For example, in the device used in this work, every time the clock frequency is adjusted, all devices fed by it (e.g., LCD controller, DMA controller, serial controllers, OS timer) 'freeze' for a duration of $150\mu s$ and the subsequent synchronization of memory requires up to $20ms$. It is to expect that newer devices will reduce these overheads, however, inefficient energy management approaches can lead to a large number of frequency adjustments, e.g., a process running for 500ms with run-time evaluations every 10ms could potentially experience 50 frequency adjustments during its execution. Instead, the goal should be to minimize the energy and time penalties caused by frequency adjustments, to maximize the number of process deadlines met, and to maximize the energy savings achieved. Simulations or models used in previous research fail to capture these significant overheads of 'real' hardware, therefore, in this paper we perform actual measurements on a handheld device to capture the overheads associated with dynamic frequency scaling. To control the overheads, we replace periodic checkpoints with an approach that *iteratively* computes checkpoints based on the *best-case execution time (BCET)*, *average-case execution time (ACET)*, and *worst-case execution time (WCET)* of a real-time application. These times can be obtained

through compile-time code analysis, or through off-line or on-line profiling. For simplicity, we can estimate the average case with $ACET = (WCET+BCET)/2$. At each checkpoint, the application progress is evaluated, a new clock frequency or voltage is calculated and set if required, and a new checkpoint is computed. This reduces the number of checkpoints and potential speed or voltage changes, e.g., our results show that the number of frequency changes is reduced to about a quarter for the experimental scenario used in this paper. This approach assumes an embedded real-time system, where tasks execute until completion (e.g., using an EDF scheduler). The approach introduced in this paper is evaluated with an application from the scientific visualization domain. An embedded device receives visualization data in form of points and lines that are to be displayed. Using profiling we derive a relationship between the number of lines in an image and the application run-time for the best-, average-, and worst-case scenarios.

2 Dynamic Frequency Scaling for Real-Time Applications

Dynamic voltage and frequency scaling (DVFS) has been introduced to trade off system performance (i.e., application execution time) with energy consumption. While this paper focuses on frequency scaling, the approach introduced here is similarly applicable to devices with voltage scaling capabilities. The processor under consideration in this paper is a StrongARM SA1110 processor and the device used in this work is a Compaq iPAQ H3870 handheld with 32MB RAM, 32MB Flash, and an Orinoco Gold 11Mbps wireless card. The processor supports 11 clock frequencies ranging from 59MHz to 206.4MHz in 14.7MHz steps, the default frequency being 206.4MHz. The device runs the *familiar* Linux distribution version 0.7.1 with a 2.4.19 kernel. Figure 1(a) compares the application

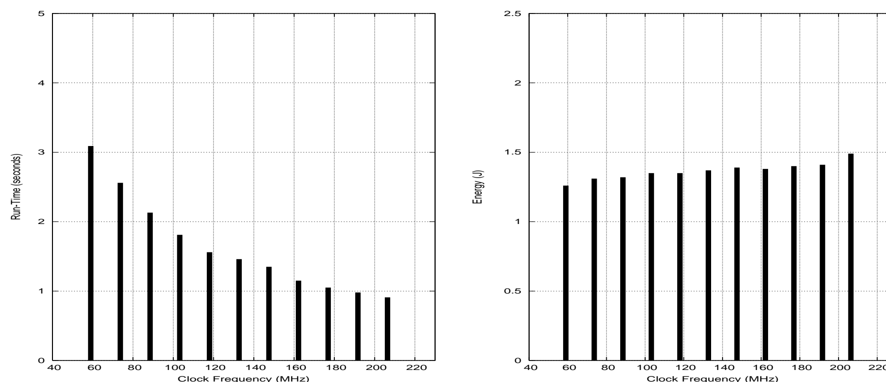

Fig. 1. (a) Application run-time and (b) device energy consumption as a function of clock frequency.

run-time of a simple test application (i.e., a *for*-loop with 10^7 iterations) at 11 different clock frequencies, showing how the application run-time increases with lower frequencies. In contrast, Figure 1(b) shows the energy consumption $E(Joule) = P_{active} * T_{active} + P_{idle} * T_{idle}$ of the device for the same application, where the shown energy is the sum of the 'active' period of the device (i.e., when an application is executed) and the 'inactive' or 'idle' period of the device over a period of $3.09s$ (the execution time of the application at the lowest clock frequency). For real-time applications it is important to select a clock frequency that allows these applications to meet their deadlines. However, uncertainties in application run-times (e.g., caused by variations in input data, the number of interrupts, etc.) would require that clock frequencies are selected such that all applications can meet their deadlines even for their worst-case execution times. However, this pessimistic approach will not fully exploit the potential energy savings, particularly if average-case and worst-case executions vary greatly. Other approaches, therefore, use dynamic evaluation of an application's progress and adjust the clock frequency if required, e.g., to speed up if the application is at risk of missing its deadline or to slow down to ensure optimal energy savings if an application is 'faster' than expected. Approaches such as profiling and compile-time analysis [1,3,16] are used to predict and monitor the run-time of an application. In [1, 3], the authors use checkpoints or hints at certain code locations to estimate the remaining execution time. However, frequent checkpoints can result in significant overheads, caused by the frequent progress evaluation and by the frequency changes. The goal of this paper is therefore to minimize the overheads by delaying progress evaluations and frequency changes until the latest possible times. Figure 2 compares the original periodic approach with the iterative approach introduced this work. In the original approach, checkpoints are placed at

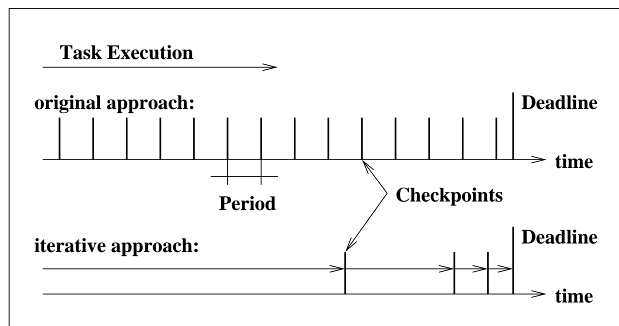

Fig. 2. Progress evaluation with checkpoints.

regular intervals, where at each checkpoint it is decided if and how to change the clock frequency. In contrast, an iterative approach uses knowledge of best-case and worst-case execution times to determine the latest possible time for progress

evaluation. At this point, the clock frequency can be adjusted if required and a new checkpoint, based on the remaining best- and worst-case execution times, is calculated. The idea is that early progress evaluations (i.e., before the location of the checkpoint computed in our approach) are unnecessary and only cause overheads through frequent progress evaluations and frequency adjustments. For example, variations in run-time detected by early checkpoints could result in frequency changes that have to be reversed later on because of other variations. Further, the accuracy of progress evaluation and frequency adjustments depend on the accuracy of checkpoint placement, i.e., an error in checkpoint placement could result in erroneous progress evaluations and undesired frequency switches. With the iterative approach, the number of checkpoints are significantly reduced, thereby reducing the negative effects of inaccuracies in progress feedback.

3 Iterative Checkpoint Computation

Assumptions and definitions. The basis of our approach is the knowledge of the *best-case execution time (BCET)* and the *worst-case execution time (WCET)* of a given real-time application. Approaches to obtain these numbers include compile-time code analysis and profiling; the latter being used in this paper. Further, the *average-case execution time (ACET)* of an application is used to compute an appropriate clock frequency. $ACET$ can be obtained in the same manner BCET and WCET are obtained, however, for simplicity, we assume that $ACET$ is the arithmetic mean, i.e., $ACET = (BCET + WCET)/2$. The maximum deviation from the mean is then $(WCET - BCET)/2$, which we denote as Δt. We assume that an application deadline T_d is either expressed explicitly (e.g., by the application) or derived from the application context, e.g., from the replay rate of a video player. The processor supports multiple clock frequencies in the range from f_{min} to f_{max}; through off-line measurements we can obtain a list of *scaling factors* $k_{n:m}$ to translate application run-times at one clock frequency to application run-times at any other clock frequency. These scaling factors are obtained by executing a sample application at all available clock frequencies and measuring the run-times, i.e., a scaling factor expresses the ratio of the run-times at two different clock frequencies. For example, an application run-time of $2s$ at f_4 and a scaling factor $k_{4:2} = 2$ translates into a run-time of $2 * 2s = 4s$ at frequency f_2. In the remainder of this document, if not otherwise indicated, all base times are assumed to be calculated for the default clock frequency f_{max}.
Frequency computation. The goal is to execute a given task P_i with a known deadline T_d at the lowest possible frequency, to allow it to approach the deadline as close as possible without missing it. The basis of our computations is the average case, i.e., we determine the clock frequency to prolong application execution assuming that the application will require the average-case execution time (see Figure 3). Therefore, the frequency f_x is determined such that the following requirements are satisfied: (A) $ACET_{max} * k_{max:x} <= T_d$ and (B) $ACET_{max} * k_{max:x-1} > T_d$. (A) determines that the average execution time multiplied by the scaling factor $k_{max:x}$ (for the transition from f_{max} to f_x) is

at most the deadline and (B) ensures that the selected frequency is the lowest possible frequency that would not cause the application to miss its deadline, assuming that the actual run-time will be $ACET$. Since the clock frequency can

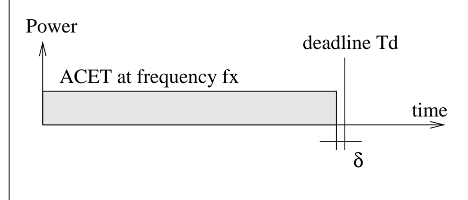

Fig. 3. Frequency selection for ACET.

only be selected at discrete steps, the application run-time $ACET_x$ ($ACET$ at the clock frequency f_x) will result in the application finishing δ time units before the deadline T_d ($\delta >= 0$). Algorithm 1 summarizes the frequency computa-

```
index = maxindex;
while (index > 0) do
    if (ACET * factor[index] <= Td) then
        index − −;
    else
        if (index < maxindex) then
            index + +;
        end
        break;
    end
end
set_clock(frequency[index]);
return frequency[index];
```

Algorithm 1: Frequency computation.

tion, where available clock frequencies and scaling factors are stored in tables ('frequency' and 'factor'). The value of *index* indicates the currently chosen table entries and thereby the clock frequency ($f_{min} <= f_{index} <= f_{max}$). The algorithm is implemented as a function in a C library, which is linked by the application. This algorithm is executed at the beginning of application execution, where the application passes the predicted average-case execution time and the deadline as parameters. The outcome of this algorithm is the selected clock frequency and the CPU clock is changed accordingly with the *set_clock* system call, which is caught by a kernel-loadable module that performs the OS-level

clock management tasks.

Checkpoint computation. Consider Figure 4, which shows the run-times of process P_i for both the best and the worst case, resulting in the task finishing $\Delta t + \delta$ time units before the deadline (best case) or $\Delta t - \delta$ time units after the deadline (worst case). Since these are the both extremes (shortest and longest paths through the application code), two checkpoints can be computed for these two scenarios and the earlier one will be the first checkpoint for the evaluation of the application progress. The remainder of this section shows how the best- and worst-case execution times are used to determine the first checkpoint.

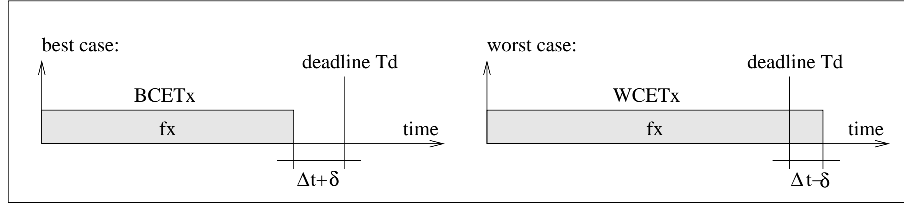

Fig. 4. Best case and worst case task run-times.

(a) Best case: In the best case scenario, an application will require $BCET_x$ at clock frequency f_x. Figure 5(a) shows the desired approach, i.e., the task is executed as long as possible at frequency f_x (for T_x time units) and at a certain – yet to be determined – checkpoint C_1, the frequency is switched to f_{min}, allowing the application to finish as close as possible to the deadline, i.e., $T_x + T_{min} = T_d$. That means that – in the case the application requires $BCET$

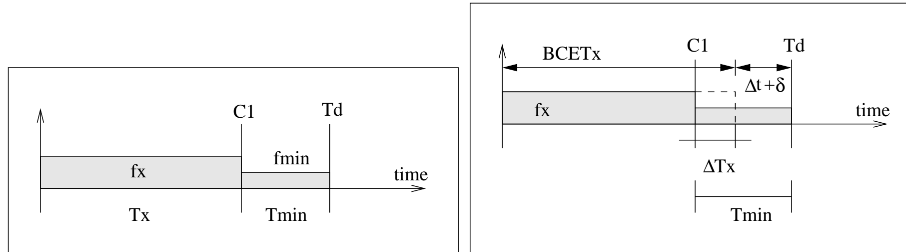

Fig. 5. (a) Desired outcome of frequency selection for best case scenario and (b) checkpoint computation.

– the first part of the task execution will occur at frequency f_x (based on the assumption the application will require $ACET$) and the remainder will occur at f_{min}, the lowest possible clock frequency. Figure 5(b) shows that scenario,

where ΔT_x is an unknown part of the task run-time $BCET_x$, which, if executed at f_{min} will satisfy the following equation:

$$T_{min} = \Delta T_x * k_{x:min} = \Delta t + \delta + \Delta T_x.$$

This leads us further to the following formula:

$$\Delta T_x = (\Delta t + \delta)/(k_{x:min} - 1).$$

Then the first checkpoint (when to evaluate a task's progress for the first time) is computed with:

$$C_1 = BCET_x - \Delta T_x.$$

(b) Worst case: In the worst case scenario, the task finishes after $WCET_x$; Figure 6(a) shows the desired approach for this case. The task is executed at clock

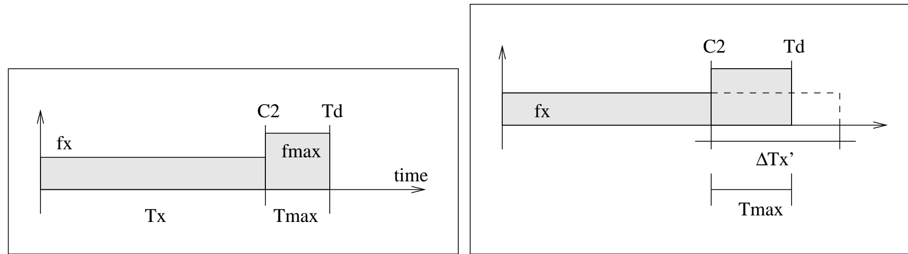

Fig. 6. (a) Desired outcome of frequency selection for the worst case scenario and (b) checkpoint computation.

frequency f_x as long as possible (for T_x time units) and at a certain checkpoint C_2 (yet to be determined), the frequency is switched to f_{max} and the remainder of the task will require T_{max} time units, where $T_x + T_{max} = T_d$. Again, this means that a certain part of the task that would be executed at f_x ($\Delta T'_x$) now has to be executed at f_{max} (see Figure 6(b)). We can establish that

$$T_{max} = \Delta T'_x * k_{x:max} = \Delta T'_x - \Delta t + \delta.$$

Or in words: an unknown time $\Delta T'_x$ of task T_i's execution (measured at clock frequency f_x) will be executed at the highest possible clock frequency f_{max}, such that the deadline will be reached exactly (in the worst-case scenario). This leads us to the following formula:

$$\Delta T'_x = (\delta - \Delta t)/(k_{x:max} - 1).$$

The checkpoint is then computed as follows:

$$C_2 = WCET_x - \Delta T'_x.$$

$\delta = deadline - (ACET * factor[index]);$
$kx_{min} = factor[maxindex]/factor[index];$
$kx_{max} = factor[index]/(1 - factor[maxindex]);$
$\Delta t_1 = ACET - BCET;$
$\Delta t_2 = WCET - ACET;$
$dtx_1 = (\Delta t_1 - \delta)/(kx_{min} - 1);$
$dtx_2 = kx_{max} * (\delta - \Delta t_2);$
$C = MIN(BCET - dtx_1, WCET - dtx_2);$
set_OS_timer(C);
return C;

Algorithm 2: Checkpoint computation.

The earliest of these two checkpoints is now the checkpoint C where the application progress will be evaluated for the first time:

$$C = min(C_1, C_2).$$

Algorithm 2 summarizes the checkpoint computation, which is performed after the clock frequency f_x is determined. This function issues a system call, *set_OS_timer*, which sets an interrupt service routine in the kernel that will be executed once the timer expires. At timer expiration, an upcall into the library is performed, where the application progress is then evaluated.

Progress counter. Each checkpoint is used to evaluate an application's progress. In addition to checkpoints, a compile-time tool must also insert frequent *progress hints* into application code, to allow checkpoints to make predictions on the remaining run-time. The goal of this paper is to minimize the costly checkpoints and frequency adjustments, assuming that the 'hinting' costs are negligible. At compile-time, two different functions are inserted into the application code: (a) a single call to the function *init_progress_counter* at the beginning of the application, initializing a *progress counter* to a pre-determined value and (b) periodic calls to a function called *update_progress*, where each call will decrement the previously initialized value of the progress counter. The initial value of the progress counter and the positions of the function calls are determined such that the progress counter can give sufficient information of the status of the application execution. At each checkpoint, the progress counter is used to evaluate the application's progress. Different approaches for the placement of calls to *update_progress* can be used, e.g., in [1], the authors use code analysis to identify optimal locations of such calls. Here, calls are placed a few milliseconds apart, details about this approach are beyond the topic of this paper. The cost of each call is a few extra instructions for decrementing an integer value.

Iterative checkpoint computation. Once the first checkpoint has been computed, the process is repeated in the following manner: we consider the remainder of a task's execution from the first checkpoint to the deadline. The previously obtained execution times for the best-, average-, and worst-case scenarios are adjusted to reflect the progress made and the new values are used to determine

(a) if the clock frequency needs to be adjusted and (b) the location of the next checkpoint. Algorithm 3 summarizes these steps, which are repeated until ei-

```
BCET_new = BCET_old * counter_rem/counter_start;
ACET_new = ACET_old * counter_rem/counter_start;
WCET_new = WCET_old * counter_rem/counter_start;
Td_new = Td_old − current_time;
dt = Td_original/P_counter;
f_y = call(algorithm 1);
C_new = call(algorithm 2);
if (C_new < 2 * dt) then
  | C_new = dt;
end
if ((C_new + 2 * dt) > Td) then
  | C_new = 0;
end
set_clock(f_y);
if (C_new > 0) then
  | set_OS_timer(C_new);
end
return C_new;
```

Algorithm 3: Recursive frequency and checkpoint computation.

ther the value of the new checkpoint or the distance between the deadline and the new checkpoint are smaller than twice dt, the interval between two consecutive calls to the function decrementing the progress counter. In the first case, instead of computing new checkpoints, we simply evaluate the progress once per dt and in the latter case we are sufficiently close to the deadline to terminate the evaluation process.

4 Evaluation

Application profiling. The application under consideration is that of a scientific visualization tool for mobile devices. Here, data streams contain graphic objects in form of points and lines, which are then displayed and the pixels between end-points of a line are interpolated. The number of lines determines the overheads in displaying the images. Each data frame received contains a header indicating the number of lines in the frame. The relationship between the number of lines and the application run-times has been determined with off-line profiling and is shown in Figure 7(a). Using these results we are able to obtain functions (e.g., with linear regression) to describe the run-times for the best case (rt_{bc}), the average case (rt_{ac}), and the worst case (rt_{wc}), where the number of lines is indicated by $num(l)$: $rt_{bc} = 2.82 * num(l) + 650, rt_{ac} = 2.86 * num(l) + 665, rt_{wc} = 2.90 * num(l) + 680$.

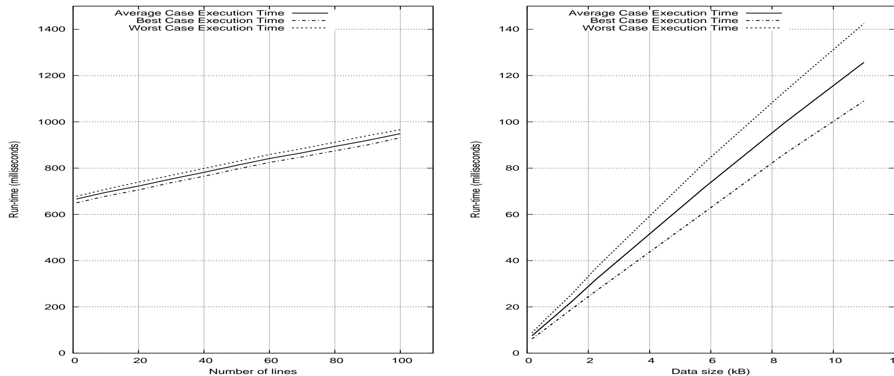

Fig. 7. (a) Scientific visualization application and (b) video decompression.

These results show that $\Delta t = (WCET - BCET)/2$ is almost independent from the number of lines. We compare these results to the profiling results of another application, a video decoder, shown in Figure 7(b). Here, the deviations Δt increase with the size of the data. This increase can be explained with the larger variation in data content for video streams, i.e., the decompression depends not only on the data size but also on the image content. The resulting functions are as follows, where $size(d)$ is the size of the compressed data in kBytes: $rt_{bc} = 9.54 * size(d) + 4.6, rt_{ac} = 10.98 * size(d) + 8.1, rt_{wc} = 12.42 * size(d) + 8.4$.

The experiments are performed on the previously described Compaq iPAQ handheld and the power measurements are performed with a Picotech ADC-100 PC oscilloscope (100kSamples/s, 2 channels, and 12 bit resolution).

Iterative checkpoint computation. The goal of the iterative checkpoint computation approach introduced in this paper is to reduce the number of progress evaluations of a running task and the number of frequency changes. Figure 8(a) shows a snapshot of an approach with periodic progress evaluation, with a period of 20ms between evaluations. This period has been carefully chosen, i.e., larger periods resulted in less frequency adjustments but more missed deadlines due to the reduced ability for the approach to react to variations, particularly close to the deadline. Smaller periods have resulted in larger overheads and more frequency adjustments and therefore also missed the deadlines more frequently. The arrows in Figure 8(a) indicate the times where the evaluation resulted in a clock frequency adjustment, e.g., 10 times in the example shown here. Further, due to the overheads caused by these frequent – and often unnecessary – adjustments, the deadline (as indicated in the graph) is ultimately missed by about 30ms. In contrast, Figure 8(b) shows a snapshot of the same application, this time with the iterative checkpoint computation. The first checkpoint is after about 30% of task execution, the second one after 50%, etc., resulting in 5 checkpoints, each of which results in a frequency change. Compared to Figure 8(a), where 20% of all checkpoints result in frequency changes (50 checkpoints and 10

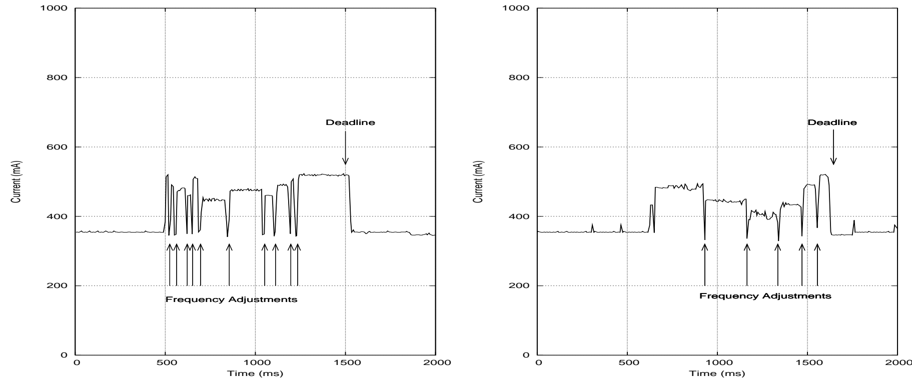

Fig. 8. (a) Current drawn by device for periodic progress evaluation and (b) current drawn by device for iterative progress evaluation.

frequency adjustments), in our approach in Figure 8(b) 100% of all checkpoints result in frequency changes (5 checkpoints), while still ensuring that the application meets its deadline. That is, our approach is efficient in the sense that only 10% of the checkpoints – compared to the periodic approach – were needed, but *each* checkpoint resulted in a frequency adjustment.

Overhead considerations. With our code, each checkpoint computation requires approximately 50μs. In contrast, each frequency switch requires about 150μs, however, the total measured delays can reach up to $20ms$. This is due to the way the used Linux version updates the SDRAM refresh rates for each frequency change. Figure 9(a) evaluates the achieved run-times of the scientific

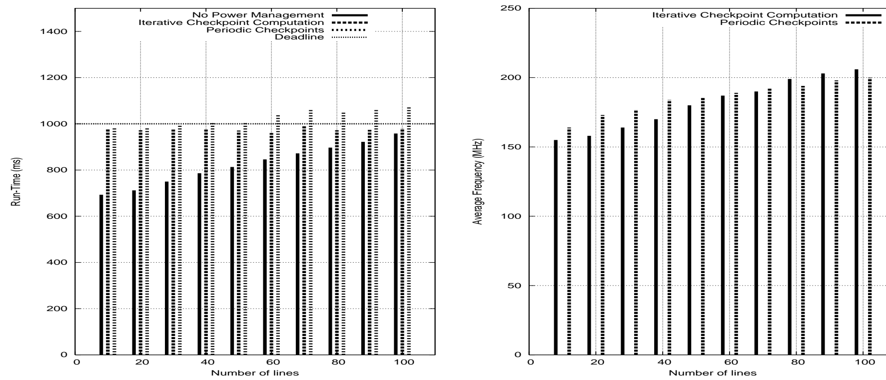

Fig. 9. (a) Measured application run-times and (b) average clock frequencies.

visualization application for the following three approaches: (a) no power management, (b) periodic checkpoints, and (c) iterative checkpoint computation. The number of lines is varied from 10 to 100, in all cases the application terminates before the deadline if no power management is deployed. If the iterative approach is used, the actual run-time is about 0-4% earlier than the deadline. However, if the periodic approach is used, the deadline is missed for most executions with number of lines of 40 and more. The reason is that the high number of evaluations and clock changes results in large overheads, which can ultimately push the execution time beyond the deadline. Figure 9(b) compares the average clock frequency for the two approaches: iterative checkpoints and periodic evaluations. Lower average clock frequency translates to lower energy consumption. For line numbers from 10 to 70, the iterative approach has lower average clock frequencies, thereby saving more energy than the periodic approach. After 70, the periodic approach has lower average clock frequencies, but note that this led to the missed deadlines in the previous graph. Similar results are shown by

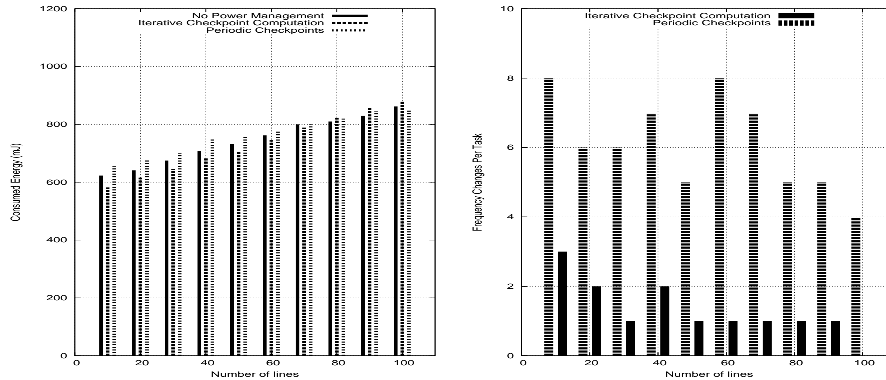

Fig. 10. (a) Energy consumptions and (b) number of frequency adjustments per application.

Figure 10(a) that show the energy consumptions as a function of the number of lines. Here, the periodic approach shows the worst results. The iterative approach shows the best results up to about 70 lines. After that, the execution times are so large and the potential energy savings so little, that the overheads caused by our approach result in increased energy consumptions compared to the case without power management. This indicates the possibility of deploying a hybrid approach, where frequency scaling is only used *if* the potential energy savings outweigh the overheads introduced by the frequency scaling algorithms (this approach is left as future work). Finally, Figure 10(b) compares the number of frequency adjustments for the same application for both the periodic and the

iterative approach at different numbers of lines, where the periodic approach requires on average 3.8 times as many clock adjustments.

5 Related Work

There has been substantial work on power management for mobile devices, including low-power modes for disks and networks [4, 6], power-aware scheduling policies [11], and energy management techniques for wireless communication [2,12]. Frequency scaling [9] and voltage scaling [10] have been investigated in recent research. Both have been shown to be useful to reduce power consumption for a variety of application scenarios, including real-time systems [7, 10]. In [15], the authors exploit slack times to integrate fixed priority scheduling with power-awareness. The exploitation of idle times to preserve power in video decoding applications has been shown feasible in previous work [8,13]. The focus of this work is on overhead reductions for dynamic frequency management for embedded real-time applications. Previous approaches have introduced methods to dynamically adjust the clock frequency to allow applications to meet their deadlines while minimizing the energy consumption [1,3]. However, frequent execution of progress evaluations and frequent frequency changes reduce the utility of these approaches. This paper extends these approaches by introducing an iterative method of computing checkpoints, reducing the number of checkpoints and frequency changes required.

6 Conclusions and Future Work

The work presented in this paper builds on previous work on dynamic frequency and voltage scaling for real-time applications. Here, periodic comparison of actual task progress with predicted task progress is used to determine if and how to change the clock frequency. The problem with periodic evaluation of application progress is that frequencies may be changed too frequent, resulting in excessive overheads. The goal of this paper is to replace periodic checkpoints with iteratively computed checkpoints, resulting in less overheads for progress evaluation and frequency changes. The results show that in the case of a scientific visualization tool, the overheads can be reduced to 26% of the costs for the original periodic approach. Our future work will use techniques to evaluate task progress also to detect ‘hot spots’ of long running application code in terms of energy consumption. For example, the scientific visualization tool discussed in this paper is typically run for a long period of time and code optimizations of frequently executed segments of the application may help to reduce the overall energy consumption. Further, this paper considered a single real-time application for embedded systems, our future work will address situations with multiple real-time applications executing simultaneously. Finally, other architectures (e.g., Transmeta, XScale) offer different numbers of frequency levels and voltage levels; our work will explore the achievable overhead reductions on these architectures.

References

1. N. Aboughazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven. Energy Management for Real-Time Embedded Applications With Compiler Support. In *Proc. of Languages, Compilers, and Tools for Embedded Systems (LCTES) Conference*, June 2003.
2. S. Agrawal and S. Singh. An Experimental Study of TCP's Energy Consumption over a Wireless Link. In *Proc. of the 4th European Personal Mobile Communications Conference*, February 2001.
3. A. Azevedo, I. Issenin, R. Comea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based Dynamic Voltage Scheduling using Program Checkpoints. In *Proc. of Design Automation and Test in Europe*, 2002.
4. S. Chandra and A. Vahdat. Application-specific Network Management for Energy-aware Streaming of Popular Multimedia Formats. In *Proc. of the USENIX Annual Technical Conference*, 2002.
5. G. Flavius. On Energy Reduction in Hard Real-Time Systems Containing Tasks with Stochastic Execution Times. In *Proc. of IEEE Workshop on Power Management for Real-Time and Embedded Systems*, 2001.
6. D. P. Helmbold, D. D. E. Long, and B. Sherrod. A Dynamic Disk Spin-down Technique for Mobile Computing. In *Proc. of the Intl. Conference on Mobile Computing and Networking*, 1996.
7. J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-Aware Scheduling under Timing Constraints for Mission-Critical Embedded Systems. In *Proc. of Design Automation Conference*, 2001.
8. M. Mesarina and Y. Turner. Reduced Energy Decoding of MPEG Streams. In *Proc. of Multimedia Computing and Networking, San Jose, CA*, 2002.
9. A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling. In *Proc. of the 16th Annual Intl. Conference on Supercomputing*, 2002.
10. P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proc. of the 18th SOSP, Chateau Lake Louise, Banff, Canada*, 2001.
11. C. Poellabauer and K. Schwan. Power-Aware Video Decoding using Real-Time Event Handlers. In *Proc. of the 5th International Workshop on Wireless Mobile Multimedia, Atlanta, GA*, September 2002.
12. C. Poellabauer and K. Schwan. Energy-Aware Media Transcoding in Wireless Systems. In *Proc. of the Second IEEE Intl. Conference on Pervasive Computing and Communications (PerCom 2004)*, March 2004.
13. J. Pouwelse, K. Langendoen, R. Lagendijk, and H. Sips. Power-Aware Video Decoding. In *Proc. of Picture Coding Symposium 2001, Seoul, Korea*, 2001.
14. S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2003.
15. Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proc. of Design Automation Conference*, 1999.
16. E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In *Proc. of the Workshop on Language, Compilers, and Tools for Embedded Systems*, 2001.
17. W. Yuan and K. Nahrstedt. A Middleware Framework Coordinating Processor/Power Resource Management for Multimedia Applications. In *Proc. of IEEE Globecom 2001, San Antonio, TX*, pages 1984–1988, 2001.