Energy Priority Scheduling for Variable Voltage Processors

Johan Pouwelse Koen Langendoen Henk Sips

Faculty of Information Technology and Systems
Delft University of Technology, The Netherlands

{pouwelse,koen,sips}@ubicom.tudelft.nl

ABSTRACT

Clock (and voltage) scheduling is an important technique to reduce energy consumption of variable-voltage processors. It is difficult, however, to achieve good results at the OS and hardware level when applications show bursty behavior. We take the approach that such applications must be made power aware and specify their future demands to a central scheduler controlling the clock speed and processor voltage. This paper describes our *energy priority scheduling* (EPS) heuristic that orders tasks according to how tight their deadlines are and how often tasks overlap. We schedule low-priority tasks first, since they can be easily preempted to accommodate for high-priority tasks later. The EPS heuristic does not always yield the optimal schedule, but has low complexity and can be used as an incremental on-line algorithm. We implemented EPS on a StrongARM-based variable-voltage platform. Measurements show that EPS reduces energy consumption with 50% for a bursty video decoding application without missing any frame deadlines.

1. INTRODUCTION

In many portable systems the microprocessor consumes a significant amount of energy. This is especially true in very small systems (PDAs) without a hard-disk and display backlight. Since battery energy is a scarce resource, energy consumption must be minimized to prolong operation time. Significant energy savings can be obtained by simply switching off the microprocessor when it is running idle; many embedded microprocessors support energy-efficient sleep modes. An even better approach is to lower the supply voltage (and clock frequency) of the microprocessor, since that amounts to a quadratic reduction ($P = C\ f\ V_{DD}^2$).

There are now several variable-voltage processors on the market that can operate reliably over a range of clock frequencies. The processor circuitry is designed such that at a lower speed it needs a lower supply voltage than at peak performance. As a consequence, the energy consumption per instruction depends on the clock speed, and varies significantly when the speed is changed. As an example, consider the Intel StrongARM 1100 processor, which is specified for 59-190 MHz and a V_{DD} of 1.5 V. We added circuitry and software to control the core voltage and are able to operate the processor from 59 MHz at 0.79 V to as fast as 251 MHz at 1.65 V (without active cooling) [11]. The difference in energy consumption per instruction is a factor of 5.

The idea of reducing (scaling) the supply voltage (and clock speed) of a processor to save energy is not new and, already in 1994, the benefits have been quantified by simulation [13]. Nevertheless, the application of voltage scaling is nearly absent in the current generation of notebooks and PDAs running a general-purpose OS. The only exception is the Transmeta LongRun solution. The main reason is that determining at what supply voltage and clock speed the processor should run to support the current workload is a difficult problem. For example, scheduling heuristics that monitor the processor load and dynamically adjust the voltage/speed accordingly, often perform worse than running the processor at a suitable fixed voltage/speed [2].

In this paper we argue that voltage scaling in a general-purpose context can only be effective when applications cooperate. It is vital that applications communicate their (future) processing needs to the OS, much like with real-time OSes. Only then the OS can handle bursty applications and compute an optimal schedule. The idea of using application-supplied workloads is also demonstrated in [6] where the power consumption of a wireless LAN is reduced. The *clock scheduling* problem itself is NP complete. We present a new heuristic scheduling algorithm called *energy priority scheduling* that uses workload descriptions to compute energy-efficient schedules. We implemented the algorithm as part of the Linux OS and performed several experiments on our variable-voltage StrongARM 1100 platform. In particular we demonstrate the ability to schedule a computational task with a bursty video playback application; the computational task is executed between two low-complexity video frames.

2. RELATED WORK

The speed at which a processor can operate is a direct consequence of the voltage supplied ($f \propto \frac{(V_{DD}-V_{th})^2}{V_{DD}}$) [4]. Therefore, determining the minimal supply voltage is equivalent to determining the minimal clock speed at which to run the processor. For simplicity we will discuss voltage scaling in terms of clock speed, but remember that a change in clock speed implies a corresponding change in supply voltage. Determining the clock speed of the processor is

ISLPED'01, August 6-7, 2001, Huntington Beach, California, USA.

not trivial. This problem is known as *clock scheduling* and consists of determining the moments when to change the speed, and to what level, given a specific workload and associated real-time constraints (i.e. task deadlines).

Clock scheduling has been investigated within three main areas: application-specific processors, real-time OSes, and general-purpose OSes. With application-specific processors, the scheduler is provided with all information beforehand. In a real-time OS accurate information is presented dynamically by the applications. A general-purpose OS, in contrast, has to derive processing demands from external measures like processor utilization. Clock scheduling becomes simpler when more (accurate) information is available.

When crafting an application-specific system the exact workload is known in advance. Therefore the *optimal* clock schedule can often be calculated with brute force at chip design time. This can be costly since clock scheduling is NP complete when tasks may not be preempted [3]; Hong et al. present an effective heuristic yielding schedules that are within 2% of the optimum [3]. In the preemptive case the optimal schedule can be computed with an $O(n \log^2 n)$ off-line algorithm [14].

Clock scheduling is more difficult in the context of real-time OSes as information only becomes available at run time. Yao et al. present their *Average Rate* run-time heuristic and prove that it computes schedules that consume at most a factor of 8 more energy than the optimal preemptive schedule [14]. Pering et al. present a heuristic based on *Earliest Deadline First* scheduling that assumes that all tasks are currently runnable [9]. Measurements show that significant energy savings can be obtained (20% of peak power) for some applications. They do not provide insight in how well their heuristic performs in comparison to the optimal schedules. Given that they do not preempt tasks, their scheduler is limited in its possibilities. Several authors [5, 12] presented algorithms that require off-line analysis of a real-time workload that is known in advance.

Weiser et al. first presented the idea of voltage scaling for a general-purpose OS [13]. Their scheduling algorithms are interval based and determine the clock speed depending on the processor utilization in previous intervals. Simulations by Pering et al. [8] show that interval-based clock scheduling on the OS-level reduces energy consumption considerably compared to running at full power. The gain, however, is extremely dependent on the interval length and the optimal setting varies per application. Moreover, applications with a bursty workload fall significantly short of optimal. For example, the energy consumption of video decoding was shown to be 36% above optimum. Evidently, the lack of information about future demands limits the effectiveness. This observation seriously questions the effectiveness of the Transmeta LongRun technology implemented in the "microcode" of the TM5400 processor. Our initial experience with the Transmeta processor shows that it exhibits an inefficient all-or-nothing behavior switching rapidly between full and minimal speed.

3. ENERGY PRIORITY SCHEDULING

At Delft University of Technology we have developed a variable-voltage testbed based on the general-purpose StrongARM processor [1]. Amongst others it will be used as a mobile computing device running Linux. We have developed a clock scheduler that combines interval-based scheduling with real-time scheduling to support both traditional applications and power-aware applications communicating their future processing demands. Requiring (bursty) applications to be cooperative is key for achieving energy efficiency as shown in the previous section. Our experience with modifying an H.263 video decoder to estimate its computational demands for each frame shows that simple measures yield accurate results [10]. Consequently, considerable energy savings are obtained (see Section 5).

		case 1	case 2
	e_j	$s_j - d_j$	$s_j - d_j$
A	2	$0-3$	$0-3$
B	2	$0-6$	$0-6$
C	1		$4-6$

Table 1: Workload descriptions.

Our *energy priority scheduler* is an incremental on-line heuristic that dynamically adjusts the clock schedule when new tasks enter the system and old tasks complete their execution. Since we are operating in a Linux environment we can safely assume that tasks may be preempted, which allows for better clock schedules. For example, when running our modified H.263 decoder in combination with some low-priority task, the latter is executed between low-complexity video frames.

3.1 Model

This section defines a model for clock scheduling. The model combines and enhances the models presented in [7] and [14]. Each real-time task j is defined by:

- s_j Starting time
- d_j Deadline time
- e_j Execution time at highest speed

The interval of task j is $[s_j, d_j]$. The energy priority scheduling algorithm is used to determine:

- $s(t)$ Speed of the processor at time t
- $run(t)$ Task that is run on the processor at time t

We further define the following parameters:

- $N_j(tr)$ Number of others tasks assigned to interval tr besides task j
- $N_j = \sum_{tr \subseteq [s_j, d_j]} \frac{N_j(tr)}{d_j - s_j}$ Average number of others tasks besides task j
- $f_j = \frac{e_j}{d_j - s_j}$ Flat processor rate of task j, uses the least amount of energy

3.2 Algorithm

Before describing our algorithm, we first present two examples that motivate the scheduling heuristic we employ. Table 1 gives two simple workloads. The first case consists of just two tasks (A and B). An incremental scheduler considers the tasks one-by-one. Following the *Average Rate* heuristic by Yao et. al. [14] we simply add the minimum required flat processor rates f_j for each task at time t. Thus, task A executes at speed 2/3 and B at speed 1/3 (see Figure 1).

The *Average Rate* schedule is not optimal since A and B can be scheduled back-to-back as shown in Figure 2. (Running at a constant speed is more energy efficient than with a varying speed).

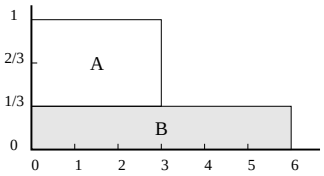

Figure 1: *Average Rate* schedule for case 1.

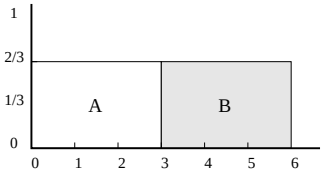

Figure 2: Optimal schedule for case 1.

A first improvement to the *Average Rate* heuristic, is to take into account the other tasks already scheduled. When scheduling a new task T we can compute the (water) level above the current schedule (contour) to fit in the computational demands (area) of T. The *task leveling* idea is outlined in Figure 3.

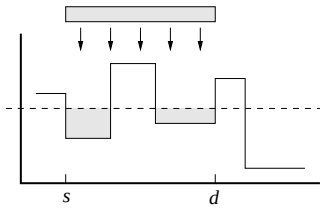

Figure 3: Task leveling.

Applying task-leveling to the first example yields the optimum (Figure 2) when scheduling task A first, followed by B. Scheduling B first and then A, however, still yields the inferior schedule shown in Figure 1.

Our second improvement is to account for overlapping tasks that can be pushed aside. Consider the second case in Table 1, which adds a third task C to the optimal schedule in Figure 2. First note that task-leveling fails to find a suitable schedule in this case since C must be layered on top of B, raising the processor utilization above 1. The following method does find the optimal schedule (an equal load of 5/6 across the entire $[0, 6]$ interval). In step one we determine the maximum processor utilization u_{max} on the interval $[s_C, d_C]$, which is 2/3 (cf. interval $[4, 6]$ in Figure 2). In step two we fill up the free space below level u_{max} on interval $[s_C, d_C]$; this has no effect in our example because there is no space available. In the third step we determine all overlapping tasks (set T) that overlap with C; T equals {B}. In the fourth step we compute the water level (5/6) above the contour of T+C that accommodates the remainder of C. Finally, we reschedule tasks T to create space in the interval $[s_C, d_C]$; see Figure 4.

Rescheduling in the final step is not always possible due to deadlines regarding tasks T, in which case steps four and five must be repeated. Dealing with overlapping tasks greatly enhances the quality of the clock schedules. Further improvements can be expected to also account for tasks that overlap with the overlapping tasks, etc. We do not pursue this direction, but rather arrange that tasks are scheduled in ascending priority. Tasks with relaxed deadlines (f_j close to 0) and few overlaps (low N_j) are ranked to be scheduled first, so they can easily be pushed aside when more difficult tasks are scheduled later.

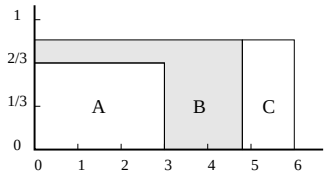

Figure 4: Optimal schedule for case 2.

Algorithm 1 Energy Priority Scheduling

0 Given a set of tasks T, each task with a starting time, deadline time, and fastest execution time.
1 Partition interval $[s_{min}, d_{max}]$ into a set of time regions $tr_i[start_i, end_i]$ where $start_i$ and end_i are start or deadline times of T, and there exists no other start or deadline time within tr_i.
2 For each task compute its priority: $p_j = f_j N_j$.
3 Repeat $\mid T \mid$ times:
3.1 Select task j that is not scheduled yet and has lowest p_j.
3.2 Repeat until task j is fully scheduled:
3.2.1 Determine intervals $tr_i \subseteq [s_j, d_j]$ with lowest scheduled processor utilization u_i
3.2.2 Determine overlapping task intervals tr_l, $tr_i \subseteq [s_l, d_l]$
3.2.3 Determine spill intervals $tr_k \in \{tr_l\} \setminus \{tr_i\}$, $u_k = u_i$
3.2.4 Define

$$\begin{aligned} u_{up} &= \text{2nd lowest processor utilization on } tr_l \\ &\quad \text{(or 1 if } \{tr_l\} \setminus \{tr_k\} = \emptyset\text{)} \\ L_i &= \sum \parallel tr_i \parallel \\ L_k &= \sum \parallel tr_k \parallel \\ \delta &= \frac{\min\{L_i u_i, L_k(u_{up} - u_i), \text{remainder}(e_j)\}}{L_k} \end{aligned}$$

3.2.5 Set processor utilizations u_l to $u_i + \delta$ and reschedule tasks (including j) on tr_l accordingly.
4 Regroup tasks spread across multiple intervals.

The details of our energy priority scheduler are presented in Algorithm 1. In steps 3.2.n a part of task j is scheduled by raising the "water" to the next level up. This level is to be found on the interval that includes all overlapping tasks. The actual increase (δ) is bound by the remainder of j that still needs to be scheduled, the amount of work that can be spilled ($L_i u_i$), and the step up ($u_{up} - u_i$). The incremental scheduling of task j in steps 3.2.n can be efficiently implemented by maintaining the overlapping intervals as a sorted list (ascending processor utilization). Once the final schedule is determined, tasks tend to be scattered over multiple intervals. To minimize the number of context switches, we regroup tasks in step 4 by swapping workloads between intervals.

The energy priority scheduling heuristic does not always find the optimal schedule, since it only accounts for pushing aside tasks that directly overlap with j. For example, when modifying case 2 slightly by changing task B to start at time 2, the insertion of task C will not raise the "water" above interval [0,2] as it could when realizing that B in turn should push task A aside. The complexity of the heuristic depends on the number of iterations needed to schedule j.

In the worst case each interval tr_i causes one step up. The maximum number of intervals is $2n-1$, leading to the upper bound of $O(n^3)$ for the complete heuristic. In practice, one or two iterations often suffice and the number of overlapping tasks is small lowering the complexity to $O(n \log n)$.

Although energy priority scheduling is presented as an off-line algorithm, it can easily be implemented as an incremental on-line algorithm. When a new task j arrives, the set of intervals tr_i must be extended followed by one round of scheduling for task j (no looping over all tasks in step 3).

4. IMPLEMENTATION

To study the effectiveness of energy priority scheduling in practice we have build a complete system from the hardware up to the application level. Figure 5 gives an overview of our system, showing the four components involved: hardware, OS, clock scheduler, and applications. All our hardware schematics and software drivers are freely available [1].

The hardware is designed around the StrongARM SA1100 processor, which supports different clock speeds: 59-251 MHz in 14.7 MHz steps. We added a variable supply voltage (DC/DC converter) that can be controlled from the CPU's general I/O pins connected to a D/A converter. We ran a number of experiments to determine the minimal voltage required at each speed setting, and added a driver to the Linux OS that uses a lookup table to select the appropriate voltage level when a new clock speed is requested. The driver additionally reconfigures the memory access timings, which are derived from the processor clock on the SA1100. The end result is that a speed/voltage switch initiated from user space completes in 140 μs, and that the power dissipated by the SA1100 shows a quadratic increase from 33 mW at 59 MHz to 696 mW at 251 MHz [11].

We implemented a clock scheduler that mediates between applications and the basic OS driver. To minimize implementation effort at the application level we designed the clock scheduler to support both unmodified applications as well as power-aware applications specifying their future needs. We use a combination of interval-based scheduling (for handling unknown workloads) and energy priority scheduling (supporting power-aware applications). We call the combined clock scheduler *PowerScale*. For convenience PowerScale is implemented as a daemon process in user space, but it can be moved inside the kernel when the need arises. An application connects to PowerScale using a UNIX socket and specifies its workload as a set of tasks with starting times, deadlines, and required cycles or minimum speed. Before running the energy priority scheduling (EPS) algorithm, PowerScale empties all sockets to consider at once all tasks currently made available by the power-aware applications. The computed schedule is then executed in a loop, listening on the socket for new tasks by invoking select() with a time-out value matching the time to the next speed change. Changing the speed involves calling our OS driver, which exports the 13 settings provided by the StrongARM; we select the setting just above the EPS speed leading to an overshoot of just 7 MHz on average. The EPS algorithm may preempt tasks. PowerScale uses the Linux process scheduler for this purpose and sends STOP and CONT signals to the processes that must be preempted.

To support traditional applications *and* to correct for miss predicted workloads PowerScale includes an extra interval-based component. By monitoring the Linux process scheduler statistics PowerScale can infer whether or not its energy-priority schedule is capable of handling the current load. When the system load (processor utilization) is close to 1, the CPU is running at the right speed. Otherwise, the speed is adjusted: an overload (util = 1) is handled by increasing the speed, an underload (util < 0.5) is handled by reducing the speed. Since we do not know what caused the system imbalance, we must employ heuristics when adjusting the speed. The relative long intervals (100 ms) prompt us to react quickly to overloads to guarantee responsiveness of the system; we double the speed increase on consecutive adjustments (exponential increase). Running at a too high speed does not impact responsiveness, only energy is wasted, and we step down to the next lower speed setting (linear decrease). The system load is also used by the energy priority scheduler to infer the average background load generated by ignorant applications. By directly accounting for this load, better schedules will be produced and less negative feedback needs to be applied by the interval-based component.

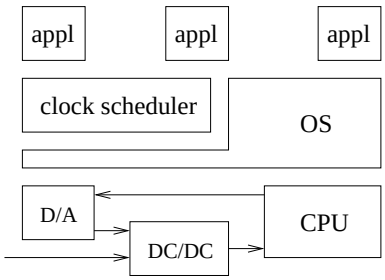

Figure 5: System overview.

Applications play an important role in achieving energy efficiency. Bursty applications like video decoding cannot be handled well by the interval-based component of PowerScale and must be modified to explicitly state their future workloads. Making applications power-aware must be considered as one of the many changes required when porting an application to a resource-constrained mobile device. As an example we adapted the Telenor H.263 video encoder to annotate the video frames with information about the decoding complexity, so that the corresponding decoder can specify the workload per frame. This fine granularity is necessary since the processing requirements for subsequent frames can differ as much as a factor of three. We found that the combination of frame type (I, P, or PB) and frame length (i.e., number of bits in the encoded stream) yields a complexity measure that is simple and accurate [10].

A second modification was required to work around the poor granularity of the internal Linux timer. The H.263 decoder has a simple rate control mechanism for displaying the frames at the specified rate (15 fps): after decoding a frame it computes the time left until the next display deadline, and invokes the usleep() system call to wait for that time to pass before outputting the video frame. Usleep() may return up to 10 ms late, which is a significant part of the frame time (67 ms). Each delay causes a frame deadline miss, and must be compensated for in the next frame to catch up. When running at a constant high speed, this happens automatically by waiting a bit shorter in the next frame. When scaling speeds, however, we must explicitly account for the inaccuracy by overestimating the computational demand of

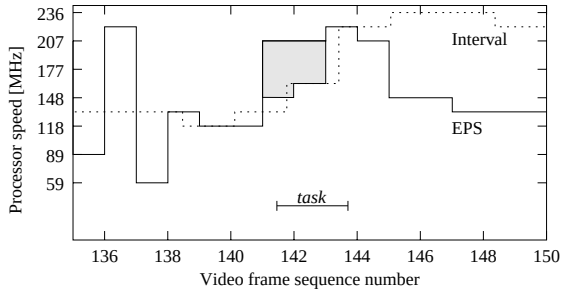

Figure 6: Clock schedules executed by PowerScale.

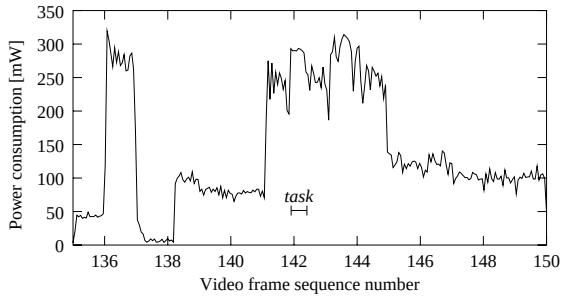

Figure 7: Processor power consumption of EPS.

each frame. We took a drastic approach and replaced the usleep() call with a busy-wait loop, in which we read the clock until the next display deadline is met.

5. RESULTS

This section reports on an experiment with our variable-voltage system. The setup involves the enhanced H.263 decoder processing the 12.6 s carphone benchmark video, encoded at a rate of 15 fps with all optimizations on. The encoded file has a size of 98 KB and is stored in main memory (RAM-disk). A second synthetic application is set to execute for a short period (150 ms, 40 MHz) near the end of the video sequence. We log the speed changes initiated by PowerScale during the experiment, and measure the energy consumption of the SA 1100 processor core by sampling (at 25 kHz) the voltage and current. Figure 6 shows the actions of the PowerScale scheduling for one second of the benchmark video (frames 135-150). The curve shows how the processor speed changes over time (each frame takes 67 ms). The corresponding power consumption of the StrongARM processor core is shown in Figure 7. For clarity this curve is down sampled by averaging uniform intervals containing 100 points.

For comparison Figure 6 shows the two modes of operation of PowerScale. The solid 'EPS' line shows the actions when PowerScale has complete knowledge (i.e. all tasks have registered their needs). The dotted 'Interval' line shows the actions when no information is available and PowerScale responds to changes in the processor load as reported by Linux. The EPS line follows the bursty workload generated by the H.263 decoder, except for the shaded area where the synthetic task is accommodated by raising the speed to 207 MHz. The interval line clearly shows that without application knowledge bursty workloads can not be handled well. The speed is either too low (e.g., frame 136) or too high (e.g, frames 144-150). One reason is that the decisions lag

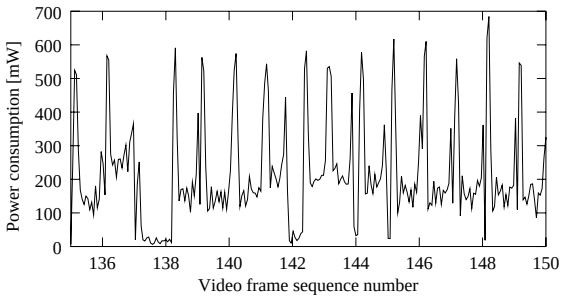

Figure 8: Total system power consumption of EPS.

behind reality because of the 100 ms scheduling resolution, for example, the increase in frame 140 is a response to the overload in frame 138; reducing the time-interval is an option, but a too short interval limits the possibilities to average out the workload at a common speed. The exact trade-off is application dependent. Another reason for the poor performance in Interval-mode is that a heuristic (exponential increase) must be applied to raise the speed up to the right level once an underload is detected leading to inefficiencies (e.g., response during frame 141 is too little). Consequently, deadlines are missed and energy is wasted.

Now consider PowerScale in EPS mode when both the H.263 decoder and synthetic application register their tasks in advance. We carefully crafted the combined workload to contain overlapping tasks. The synthetic task enters the system 25 ms after frame 141 starts and must finish 25 ms before frame 143 ends; the start-stop interval is indicated in Figure 6. The synthetic task thus overlaps with frames 141, 142, 143, and 144. The EPS algorithm schedules the synthetic task first, because it has the lowest flat processor rate (40 MHz), followed by 141 (148 MHz), 142 (162 MHz), 144 (207 MHz), and 143 (221 MHz). The final schedule raises the processor speed during the decoding of frames 141 and 142 (i.e. the shaded area in Figure 6). This effectively creates a 30 ms gap between frame 141 and 142, which contains enough cycles to run the synthetic task ($30 \times 207 > 150 \times 40$).

The measured power dissipation of the processor (Figure 7) shows a shape that is quite similar to the clock schedule executed by PowerScale in EPS-mode (Figure 6). Note, however, that the peak-to-bottom power ratio is larger than the corresponding speed ratio. Neglecting frame 137, which requires no computation and causes the processor to enter its special idle-mode, the peak-to-bottom power ratio is around 6 ($\text{frame } 136 : \text{frame } 135 \approx 271 : 43$), the speed ratio is around 2.5 ($221 : 89$). This shows the effect of the quadratic relation between power and voltage. The exact time location of the synthetic task is marked in Figure 7. Its power consumption is larger than that of its neighboring decoding tasks running at the same speed because the synthetic task does not reference main memory, hence, incurs no processor stalls when waiting for memory accesses to complete.

Figure 7 shows the power dissipated by the variable-voltage core (CPU + cache) of the StrongARM processor only. The remaining parts of the system (bus, memory, etc.) are powered from a fixed 3.3 V. Figure 8 shows the power dissipation of the 3.3 V part of our system. The high peaks correspond to the reading of the encoded frame into cache memory, which is expensive because the video is stored in FLASH memory. The system activity is similar for each frame, except for frames 136 and 137 that involve decoding a PB

frame, irrespective of the clock speed set by PowerScale. Note that the average system power (202 mW) exceeds the average processor power (118 mW), which limits the overall effectiveness of voltage scaling.

	power [mW]	misses [ms]
235 MHz	400	0
interval	337	401
EPS	304	0

Table 2: Total power dissipation and accumulated deadline misses.

Table 2 shows the average total (core + system) power dissipation over the complete carphone video. For reference we measured the power dissipation of running at a fixed 235 MHz, which allows all P and PB frames to be decoded with the 67 ms frame time. Interval-based scheduling reduces energy consumption with 16%, but at the cost of missing deadlines. For each frame we recorded the time at which it was actually written to the display buffer and accumulate the deadline misses. PowerScale in EPS mode does not miss a deadline *and* reduces energy consumption with 24%. (When considering the processor only, EPS reduces energy consumption with 50%.)

6. CONCLUSIONS AND FUTURE WORK

Clock (and voltage) scheduling is an important technique to reduce energy consumption of mobile devices equipped with a general-purpose variable-voltage processor. From the hardware perspective the gains are impressive, for example, the StrongARM SA1100 processor running at 251 MHz requires five times more energy per instruction than when running at 59 MHz. From the software perspective, however, it is difficult to achieve such reductions when applications show bursty behavior. OS-based approaches like interval scheduling do manage to reduce energy consumption somewhat, but at the expense of missing deadlines. We have shown that by requiring applications to be power aware (i.e. they must specify their future demands) much better energy reductions can be achieved while still meeting all deadlines.

This paper describes our *energy priority scheduling* (EPS) heuristic that given a set of tasks yields a clock schedule for controlling the speed (and voltage) of the processor. The approach is to order tasks according to how tight their deadlines are and how often tasks overlap with others. We schedule low-priority tasks first, since they can be easily pushed aside (preempted) to accommodate for high-priority tasks scheduled later. The heuristic does not always yield the optimal schedule, but has low complexity and can be used as an incremental on-line algorithm with little modification.

To demonstrate the effectiveness of EPS we have actually build a complete system consisting of variable-voltage hardware (StrongARM based), OS support (Linux driver), clock scheduling daemon (PowerScale), and power-aware applications (H.263 video decoder). We measured and analyzed the effectiveness of EPS with a workload consisting of the power-aware video decoder competing with a computational task. The results show that EPS successfully schedules both applications and reduces the energy consumption of the processor with 50% when compared to running at full speed (235 MHz), which is a significant improvement over interval-based scheduling achieving 33% reduction. EPS achieves this reduction without missing deadlines, unlike interval scheduling that does miss deadlines.

Currently our PowerScale daemon always selects a schedule that complies with task deadlines. In the future we would like to investigate the trade-off between energy reduction and deadline misses. Knowing this trade-off is useful when batteries are low, and the user might accept a (slight) degradation in performance in favor of a (much) more energy-efficient execution. We also would like to extend our work to include other resources such as hard-disks, wireless connections, and smart batteries such that PowerScale can determine the remaining battery life at the current performance level. Eventually, PowerScale should be capable of automatically adjusting the performance of applications (e.g., lowering the audio quality) to match a user requested lifetime.

Acknowledgements

This work was conducted within the Ubicom program (`www.ubicom.tudelft.nl`) funded by the TU Delft, DIOC research program. We thank Jan-Derk Bakker and Erik Mouw for providing us with an excellent low-power platform, and assisting us with the measurements and their interpretation. We thank Hylke van Dijk, Dick Epema, and Arjen van der Schaaf, for commenting on draft versions of this paper.

7. REFERENCES

[1] J.-D. Bakker, J. Mouw, and M. Joosen. Linux Advanced Radio Terminal. `http://www.lart.tudelft.nl/`

[2] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *OSDI*, San Diego, CA, October 2000.

[3] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable voltage core-based systems. In *36th Design Automation Conference*, pages 176–181, June 1998.

[4] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISPLED*, 1998.

[5] C. Krishna and Y.-H. Lee. Voltage-clack-scaling adaptive scheduling technique for low power in hard real-time systems. In *RTAS*, May 2000.

[6] Y.-H. Lu, L. Benini, and G. D. Micheli. Requester-aware power reduction. In *ISSS*, Sept. 2000.

[7] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *IEEE Int. Conf. on Acoustic, Speech, and Signal Processing (ICASSP'00)*, pages 3239 –3242, June 2000.

[8] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISPLED*, Aug. 1998.

[9] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *ISPLED*, 2000.

[10] J. Pouwelse, K. Langendoen, R. Lagendijk, and H. Sips. Power-aware video decoding. In *22nd Picture Coding Symposium*, Seoul, Korea, Apr. 2001.

[11] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Mobicom'01*, Rome, Italy, July 2001.

[12] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *ICCAD*, Nov. 2000.

[13] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *OSDI*, pages 13–23, 1994.

[14] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *36th IEEE Symposium on Foundations of Computer Science*, pages 374–382, 1995.