Application-Directed Voltage Scaling

Johan Pouwelse, Koen Langendoen, *Associate Member, IEEE,* and Henk J. Sips, *Associate Member, IEEE*

***Abstract*—Clock (and voltage) scheduling is an important technique to reduce the energy consumption of processors that support voltage scaling. It is difficult, however, to achieve good results using only statistics from the operating system level when applications show bursty (unpredictable) behavior. We take the approach that such applications must be made power-aware and specify their *average execution time* (AET) and the deadline to the scheduler controlling the clock speed and processor voltage. This paper describes our *energy priority scheduling* (EPS) algorithm supporting power-aware applications. EPS orders tasks according to how tight their deadlines are and how often tasks overlap. Low-priority tasks are scheduled first, since they can be easily preempted to accommodate for high-priority tasks later. The EPS algorithm does not always yield the optimal schedule, but has a low complexity. We have implemented EPS on a StrongARM-based variable-voltage platform. We conducted experiments with a modified video decoder that estimates the AET of each frame. Measurements show that application-directed voltage scaling reduces processor power consumption with 50% for the bursty video decoder without missing any frame deadlines.**

***Index Terms*—Adaptive software, low power, power awareness, voltage scaling.**

I. Introduction

POWER CONSUMPTION is becoming the *limiting factor* for the functionality of wearable devices, because advances in battery technology are progressing slowly whereas computation and communication demands are increasing rapidly. It is, therefore, important to utilize the available energy as efficient as possible. Energy preservation, or energy management, is further translated into a low-power consumption of all parts of a wearable device. The initial response to the low-power demand was to lower the supply voltage. For example, by reducing the supply voltage from standard 5.0 to 3.3 V power was reduced by 56%.

Additional reductions can be obtained by selectively lowering the supply voltage of specific parts in either a discrete or continuous manner. An obvious candidate is the processor since it is responsible for a significant portion of the total power consumption [1].

A discrete approach to voltage reduction is using power down features to minimize the power consumption of unused hardware. For portable computers this means turning off the hard disk, processor, screen, modem, sound, etc. Reactivation of hardware can take some time, which affects performance (e.g., response time). Using simple power-down-when-idle techniques the processor's power consumption can be significantly reduced. Power savings up to 66% have been reported [2].

A further refinement is to make continuous tradeoffs between performance and cost. Performance metrics are application dependent, but often a combination of response time and quality is used. Video decoding is used as a case study throughout this paper, and typical quality metrics are spatial/temporal resolution, color depth, and distortion level. The user demand (performance) should be satisfied at the lowest cost (power consumption). *Voltage scaling* is a method to tradeoff processor frequency (performance) against power consumption. The power consumption of a processor running at a high frequency and high voltage is much larger than running at a low frequency and low voltage. The power consumption of digital CMOS circuits can be modeled quite accurately by simple equations ($P = \alpha C f V_{DD}^2$) [3], [4].

A. Voltage Scaling Implementations

In 1996, one of the first papers was published that describes an actual hardware implementation using voltage scaling [5]. This implementation applies voltage scaling to MPEG video decoding on a DSP. The frequency and voltage are adjusted to match the varying complexity of video frames. In [6], a dedicated cryptography processor is presented that uses voltage scaling. When running at 50 MHz this processor requires a supply voltage of 2 V and consumes at most 75 mW; at 3 MHz, a supply voltage of only 0.7 V is required and the power consumption drops to a mere 525 μW.

In 1998, the first experimental results on a general-purpose processor were published [7]. The architecture of a R3900 RISC core was enhanced with a critical path replica to measure the minimal required supply voltage. The RISC core operates on 1.9 V at 40 MHz and on 1.3 V at 10 MHz. All intermediate frequencies are also supported. This first general purpose implementation did not have a full chip-set and lacked an operating system. In 2000, Grunwald *et al.* presented experimental results on a complete general-purpose platform, called Itsy, running the Linux operating system [8]. Itsy uses a standard commercial StrongARM SA1100 processor that supports voltage scaling. The savings by the Itsy are very modest because only two voltage levels have been implemented, 1.5 V ($\geq$162 MHz) and 1.23 V ($<$162 MHz). The resulting difference in processor power consumption between the two levels is only 15%. Better results are obtained with the SmartBadge platform, which is similar to the Itsy. Extensive power measurements on real-time MP3 audio decoding and MPEG video decoding show that an

Manuscript received February 28, 2002; revised August 27, 2002. This work was supported by the Ubicom program under Delft University of Technology, Delft Interdisciplinary Research Center (DIOC). The work of J. Pouwelse was supported by the Dutch Organization for Applied Scientific Research (TNO), Physics and Electronics Laboratory.

The authors are with the Faculty of Information Technology and Systems, Delft University of Technology, The Netherlands (e-mail: pouwelse@ubicom.tudelft.nl; koen@ubicom.tudelft.nl; sips@ubicom.tudelft.nl).

Digital Object Identifier 10.1109/TVLSI.2003.814324

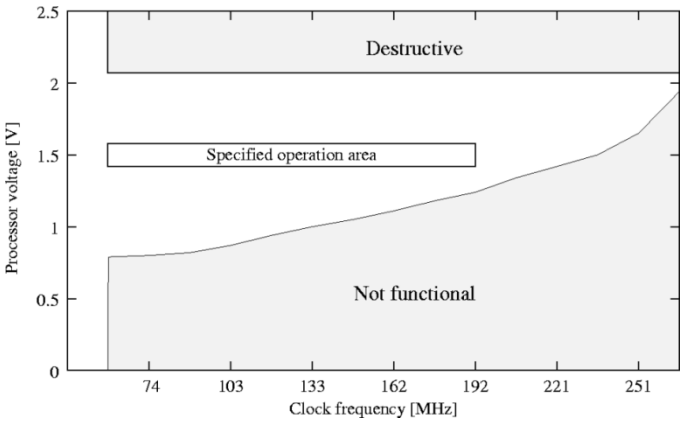


Fig. 1. Processor envelope.

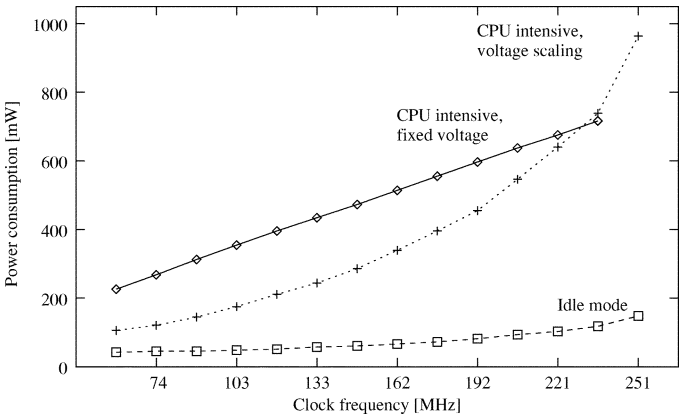


Fig. 2. Total-power consumption for idle and CPU-intensive workloads.

energy reduction of 40% is possible [9]. Burd and Pering designed and implemented a voltage-scaling capable processor based on an ARM8 core [10]. Their processor is fabricated in 600–nm technology and uses aggressive power saving features. In high-performance mode, it runs at a speed of 80 MHz and consumes 476 mW at 3.8 V. When running at 5 MHz and 1.2 V, the processor only consumes 3.24 mW. Thus, power consumption is reduced with a factor of 147, while performance drops with a factor 16.

In parallel, with the above projects we have created our own portable platform for voltage scaling research [11]. It is somewhat similar to Itsy; we also use a standard SA1100 processor and run Linux. Our platform is called LART, and described in detail in Section V-A. Fig. 1 shows the processor envelope for the different frequencies that are supported by the StrongARM processor. The LART supports 128-different supply-voltage levels. A supply voltage of 0.79 V is sufficient when running the processor at 59 MHz. A frequency of 251 MHz requires 1.65 V. These supply voltages are outside the manufacturers specifications of 1.5 V. All processors we obtained were able to run at these voltage and frequency combinations. A switch between such combinations takes 140 μs. A number of destructive tests indicated that the maximum frequency of the SA1100 processors is around 265 MHz, significantly beyond the official specified maximum of 190 MHz.

We measured the effect of voltage scaling on the power consumption of the complete LART platform, including memory, voltage conversion, etc. Fig. 2 shows the total power consumption of the LART under two different workloads: idle and CPU-intensive.

The idle workload measures the background power consumption of the LART, which is always spent regardless of the processor load. The Linux scheduler puts the processor into halt mode when no processes are active. Halt mode stalls the CPU, but other services of the embedded processor such as the memory controller and internal timer are still operational [12]. All these services are driven by the processor clock, which explains why the power consumption in halt mode increases with the frequency. The SA-1100 also supports a more power efficient sleep mode, but this mode interrupts direct memory access (DMA) transfers, stops the LCD controller, blocks memory access, etc. Also, the wake-up sequence takes much longer than in halt mode compromising responsiveness.

The CPU-intensive workload consists of the Dhrystone benchmark utilizing both the CPU and the cache. We first measured the effect of scaling the clock frequency while keeping the voltage constant at 1.5 V. In this case, the power consumption increases linearly with the frequency, as is expected. Next, we measured the power consumption when the core voltage is set to the minimal value reported in Fig. 1. The resulting curve shows the expected quadratic increase of power consumption when the frequency is varied from 59 to 251 MHz.

From the power consumption at 59 MHz (105.8 mW) and at 251 MHz (963.7 mW) it follows that an instruction at peak performance consumes a factor 2.1 more energy than at lowest performance. When we neglect the nonCPU subsystems of the LART, which are supplied from a *fixed* 3.3 V, and focus on the CPU, the power consumption is 33.1 mW at 59 MHz and 696.7 mW at 251 MHz (not shown). The raw CPU energy/instruction difference is thus, a factor 4.9.

Voltage scaling is moving from the research field into the commercial market place of embedded and x86-compatible processors. AMD has added voltage scaling capabilities to the AMD K6 processor family in April 2000. The AMD-K6-IIIE supports clock frequencies from 200 to 500 MHz, the power consumption is 2.95 and 11.40 W, respectively. This means that the lowest frequency provides a power-efficiency improvement per instruction of 55% versus the highest frequency. Transmeta and Intel currently also provide processors with voltage scaling. Due to the rising importance of power consumption it is likely that voltage scaling will soon become a standard feature for processors in the embedded and laptop market.

B. System Architecture

In this paper, we concentrate on a wearable platform consisting of a general-purpose processor with voltage-scaling capabilities, controlled by a general-purpose operating system, and running multiple applications. Fig. 3 gives an overview of such a system. The clock scheduler optimizes the processor frequency with respect to the workload to be serviced. The clock scheduler, part of the operating system (OS), must determine

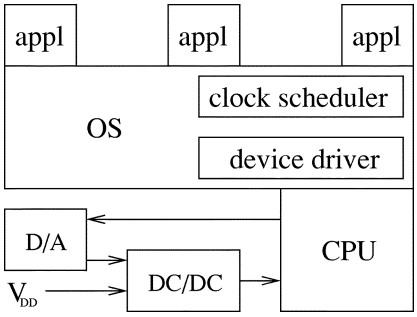


Fig. 3. System overview.

when the clock frequency needs to be changed and to what frequency. This problem, known as *clock scheduling*, is the central problem addressed in this paper. The actual switch of the processor clock frequency is handled by an OS device driver. We have implemented such a driver for the Linux OS (details can be found in [13]). The device driver adjusts the supply voltage of the processor and adapts memory and bus configurations. Note that a change in frequency implies a corresponding change in supply voltage. Our LART device driver has a hard-coded table of the frequency and voltage combinations, derived from Fig. 1. The device driver sends the voltage setting to a D/A converter. This D/A converter in turn is connected to a voltage controlled dc/dc converter. On our LART platform, the switch of frequency and voltage takes 140 μs, during which the system is stalled. This implies that settings can be changed frequently without causing too much overhead.

C. Paper Outline

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 in real-time systems. Only then the OS can handle bursty (unpredictable) applications and compute an optimal schedule. The general clock scheduling problem itself is NP complete. We present a new heuristic scheduling algorithm called *energy priority scheduling* (EPS) that uses workload descriptions to compute energy-efficient schedules. We have implemented the algorithm as part of the Linux OS and performed several experiments on our variable-voltage LART platform. In particular, we demonstrate the ability to schedule a computational task with a bursty video playback task; the computational task is executed between two low-complexity video frames.

Section II presents a general framework for the various approaches to solve the clock scheduling problem. Section III introduces the concept of power-aware applications and describes how we have added power-awareness to a H.263 video decoder. Section IV describes our energy priority scheduling algorithm. In Section V, we discuss the implementation of the EPS algorithm and present power measurements. Finally, in Section VI we conclude and indicate future research directions.

II. Approaches and Related Work

Voltage scaling and clock scheduling have been investigated in the context of four main areas: dedicated hardware, compilers, real-time OSs, and general-purpose OSs. They differ in the time of fixation of the clock schedule, the time at which scheduling information is available, and the amount and quality of that information. With dedicated hardware the clock schedule is determined at design time, using *a priori* information derived from the application. A compiler, in contrast, can only extract a limited amount of information from the source code of an application to determine the clock schedule. A real-time OS includes a task scheduler that takes into account start times, deadlines, and required cycles, allowing more flexible clock scheduling schemes. A general-purpose OS has to derive a clock schedule from run-time statistics, such as the processor utilization in previous periods. In general, clock scheduling becomes simpler and more power efficient when more (accurate) information is available. An overview of the different approaches is shown in Table I. The first row lists the quality of the information that is available to solve the clock scheduling problem, ranging from very poor $(--)$ to very good $(++)$. The second row lists at what time workload information is available: during design time, compile time, or at run time. In the sequel, we will use the approaches in Table I to discuss related work.

A. Dedicated Hardware

When crafting dedicated hardware, for example, a global system for mobile communication (GSM) speech codec or JPEG compressor, all possible workload details are known in advance. Therefore, the *optimal* clock schedule can often be calculated with brute force at chip-design time [14]. This can be costly since the nonpreemptive clock scheduling problem, where task cannot be interrupted, is NP complete [15]; Hong *et al.* present an effective heuristic yielding schedules that are within 2% of the optimum [15]. In the preemptive case, the optimal schedule can be computed with an $O(n \log^2 n)$ offline algorithm [16].

B. Compiler

When a compiler is used to determine the clock schedule, the largest problem is to deduce the appropriate information from the source code. For example, deriving the execution time on the target platform from the high-level program code is a nontrivial task. This forces the compiler to make conservative assumptions and yields low-quality scheduling information. If extensive profiling information is present, the scheduling techniques for dedicated hardware can be used. Otherwise, heuristics must be applied to identify code sections that can be executed at low speeds. For example, Hsu *et al.* describe a system that is based on identifying memory-bound loops [17]. Within such loops the clock frequency can be reduced, since the memory subsystem is much slower than the processor. This approach, however, can only be effective when memory-bound loops occur frequently and the cost of a frequency/voltage change is negligible relative to the total execution time of a loop.

C. Real-Time OS

In the realm of real-time OSs, voltage scaling focuses on minimizing power consumption of the system, while still meeting strict task deadlines. Real-time tasks specify their

TABLE I
COMPARISON OF CLOCK SCHEDULING APPROACHES IN SEVERAL AREAS

	dedicated hardware	compiler	real-time OS		general-purpose OS			application directed
			fixed	dynamic	hardware	interval	integrated	
Quality	++	–	+/-	+	–	-	+/-	++
Availability	design	compile	design	run	run	run	run	run

starting time and deadline; tasks that must be repeated also specify their period. In hard real-time systems, the worst case execution time (WCET) can often be obtained at software design time through static analysis, profiling, or direct measurements [18], [19]. When all details of the workload are known and the schedulability is verified at design time we classify such systems under "fixed real time." When details of the workload, such as WCET or even the tasks themselves, are only known at run time, we classify such systems under "dynamic real time." For example, for multimedia servers the exact workload is only available at run time [20]. An admission controller needs to determine if new tasks can be scheduled and admitted.

An example of a scheduler for fixed real-time systems is the *average rate* run-time heuristic by Yao *et al.*, which is proved to consume at most a factor of 8 more energy than the optimal preemptive schedule [16]. Pering *et al.* present a dynamic real-time system based on *earliest deadline first* (EDF) scheduling [21]. They assume that tasks specify no start times and, hence, can be executed at any moment. Measurements show that significant energy savings can be obtained (20% of peak power) for some applications.

In both classes of real-time OSs, the WCET is used to check the schedulability and possibilities for reducing the clock frequency in the schedule without violating deadlines. For example, the algorithm in [15] initially schedules all tasks at maximum frequency. After that the task schedule is adjusted until no further reduction is possible without violating deadlines. The ratio between the actual execution time and the WCET can be quite low: an average of 0.5 is reported for several hard real-time applications studied in [22]. When the WCET is not an accurate estimation of the execution time, the assigned clock frequencies to meet deadlines tend to be too high (factor of 2 on average). Consequently, a task usually finishes early, and an idle periods occurs. If another task is eligible for execution, however, the idle period can be used to execute that task at a reduced speed, see [23].

A recent paper by Pillai *et al.* discusses an alternative approach to handle the conservative WCET rarely encountered in practice [24]. Their "look-ahead" clock scheduling algorithm is based on an EDF scheduler. The task with the earliest deadline is scheduled with the lowest possible processor speed that does not violate its deadline. This forces other tasks to be scheduled at a high frequency to compensate. The assumption is that the task is not likely to use its WCET and will finish early. When the task finishes early, energy is saved and the next task with the earliest deadline is scheduled at its lowest possible frequency. Note that for some (artificial) workloads the look-ahead algorithm may defer tasks too aggressively and actually increase power consumption, as can be derived from their simulation results.

D. General-Purpose OS

Clock scheduling in the context of a general-purpose OS is difficult, since little information is known about the applications. Applications do not communicate deadlines or priorities to the OS, hence, all the clock scheduler can do is observe the load that has been generated in the past and extrapolate into the future. The clock scheduler measures the processor load in fixed intervals, for example, every 20 ms. A common technique is to use two boundary values on the processor load to decide whether to increase, decrease, or keep the current clock frequency in the next interval. If the measured processor load drops below the lower bound, the processor frequency is decreased. Similarly, if the processor load rises above the upper bound, the frequency is increased. This technique is called interval-based clock scheduling, or interval-scheduling for short.

The Transmeta Crusoe processor is the prime example of a hardware-based approach to interval-scheduling. It has built-in support for clock scheduling in the "microcode" of the processor [25]. Unfortunately, little information is made available about the exact workings of the "LongRun" technology, but it is clear that it can operate in isolation, that is, without any help from the OS or application [26]. The microcode has only a small awareness of the global system state, for example, it can not distinguish OS foreground tasks from background tasks.

Weiser *et al.* first presented the idea of interval-based voltage scaling for a general purpose OS in 1994 [27]. Most contributions regarding interval-based voltage scaling consist of theoretical analysis [4] and simulations [28]–[30]. The simulation studies show that interval-scheduling reduces power consumption considerably compared to running at full power. There are, however, some fundamental problems. First, the optimal interval length is application dependent. Second, bursty applications with unpredictable workloads cannot be scheduled effectively at all. The simulations by Pering *et al.* show that the power consumption of their interval schedule for video decoding was 36% above the optimum. Recent measurements on actual hardware by Grunwald *et al.* confirm these observations [8].

Traditional interval scheduling based on processor load can be improved by incorporating other information (run-time statistics) to estimate the processing requirements of applications. Such "integrated clock schedulers" require numerous modifications to the OS. For example, Flautner *et al.* [31] describe

an integrated scheduler that maintains processor usage statistics of every process, observes the communication pattern between processes, keeps track of input/output device usage by processes, and tries to extract deadlines from periodic tasks. The simulations results are promising, but a comparison with a traditional interval-scheduler is not included, so the advantage of using additional information is still to be determined. Another aspect that needs additional study, is the effect of their scheduler on bursty and CPU-intensive applications, such as video decoding and speech recognition, because such applications were not included in the simulations.

User related timing characteristics are another useful source of information for integrated clock scheduling [31], [32]. For example, Lorch *et al.* [32] exploit the observation that a reaction time of 50 ms for interactive applications is below the perception threshold of the user. Therefore, the application processing time for, say, a mouse click can be increased to 50 ms (by slowing down the CPU) without noticeable performance degradation. Offline simulations show that the upper bound on the additional energy saving is in the order of 20%. It remains to be seen how much energy can actually be saved in a real implementation.

E. Application-Directed Clock Scheduling

Reliable, accurate information for solving the clock scheduling problem can only be obtained from the applications themselves. When applications operating on a general-purpose OS are modified to register their processing requirements (cycles, deadlines, etc.) clock scheduling becomes simpler and more effective. Application-directed clock scheduling holds two opportunities for further power savings compared with the integrated clock scheduler. The first opportunity is to allow the updating of processing requirements. The second opportunity is to use average processing requirements instead of the worst case estimates that are used in real-time systems.

We call the updating of task processing requirements in application-directed scheduling *intratask information updates*. Intratask information updates are proposed in a recent paper by Shin *et al.* [33]. They combine the compiler-based approach with application-directed clock scheduling. Shin *et al.* use source code analysis to extract the WCET and combine it with a run-time component. An MPEG 4 decoder is used as a case study for their analysis tool. The tool calculates offline the WCET updates for several points in the MPEG 4 frame decoding process. For example, at the start of the frame decoding process only the overall worst WCET of any frame type is known. When the frame type is determined, it is replaced by the WCET of that frame type. During the decoding of the frame even more information becomes available (such as motion vectors and macroblocks) and the WCET is updated and converges to the actual frame decoding time. This refinement technique is guaranteed to meet hard real-time deadline requirements (i.e., frame deadlines). The disadvantage is that the overall WCET is not very likely to occur and consequently, the clock speed is set far too high at the beginning of every frame.

In contrast to real-time systems, applications operating on a general purpose OS are not time critical and deadlines may occasionally be missed. Typical laptop applications such as word processing, games, video editing, and (wireless) web browsing are real-time (interactive) applications, yet their deadlines are soft. Users will tolerate (some) jitter in response times. This allows for an easy solution to the problems associated with the WCET. Applications may report their average execution time (AET), which is generally a much better estimator of the true execution time, leading to more power-efficient clock schedules. This is the approach we take in this paper. Note the resemblance with the look-ahead algorithm from Pillai [24] (see Section II-C). The look-ahead algorithm is the only algorithm known to us that also exploits the low likelihood of WCET occurrences *before* the actual execution of a task.

The application-directed clock-scheduling algorithm presented in this paper can be applied in both a real-time and general-purpose OS context. The applications on a general-purpose OS need to be modified to pass on intratask information updates and indicate their AET.

III. Power-Aware Video Decoding

To exploit the power consumption reduction of voltage scaling, we propose to make applications power-aware such that bursty and cpu-intensive applications can decrease their power consumption by indicating their processor usage to the clock scheduler. We modified a video decoder to estimate its AET for each frame and communicate this requirement along with the frame deadline to the clock scheduler. In this section, we briefly discuss H.263 video compression, our method for estimating the frame decoding time, and our modified H.263 application.

A. H.263 Video Compression

The H.263 standard is created for low-bitrate video compression [34]. The standard is based on both H.261 and MPEG2. H.263 frames are displayed at a fixed rate. Throughout this paper we use a framerate of 15 fps, which means a maximum decoding time of 67 ms per frame. H.263 defines three types of frames: I-frames (intrapicture), P-frames (predicted picture), and B-frames (bidirectional predicted picture). I-frames are self contained images, similar to JPEG. P-frames encode the difference from a previous I or P frame. B-frames contain references to both preceding and succeeding frames. Because a B-frame contains forward references, the succeeding frame must be decoded prior to the B-frame itself. As a result the decoder must process two frames in a single frame time. We use the PB-frame notation to indicate the frame in which two dependent consecutive frames are decoded.

A frame consists of a grid of blocks that measure 16×16 pixels, called macroblocks. A macroblock in a P-frame consists of the differences with a reference to the previous frame that is displaced by a vector to compensate for motion. Motion compensation is used to decrease the difference from the previous frame. The pixels in each macroblock are efficiently encoded using a discrete Cosine transform (DCT), which is a computational intensive operation. The number of bytes for each macroblock in the encoder output is variable. Macroblocks that contain no information are not inserted into the compressed bit-

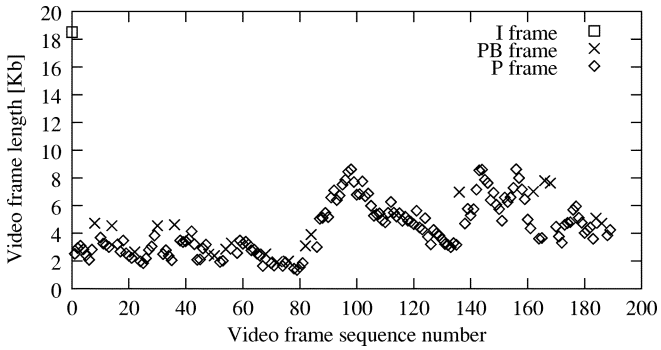


Fig. 4. Frame size variation over time.

stream. The number of inverse DCTs required to decode the macroblocks is, therefore, variable. This is the main cause for the bursty behavior of both H.263 and MPEG2 decoders. Variable-length encoding (e.g., Huffman coding) is used to further compress the macroblocks, the motion vectors, and the frame header. Note that the type of the frame is only known to the decoder *after* the variable-length decoding step.

B. AET Estimation

Predictions of decoding times are difficult to make due to the wide variation in scenes (e.g., talking heads versus MTV). A frame that is very similar to the previous frame results in few encoded macroblocks to capture the difference, hence, takes little time to decode. Frames that differ considerably from their predecessor result in longer decoding times. Fig. 4 shows the variation of the frame size for the well-known carphone test sequence (190 frames), which was encoded using the Telenor H.263 encoder V2.0 with the following settings: qcif resolution, 15 fps, default quantization, unrestricted motion vectors, syntax-based arithmetic coding, advanced prediction mode, and use of PB-frames. Note the large initial I-frame in the upper left corner.

Various methods have been developed to estimate the AET of a video frame. One method is to include a complete reference decoder inside the encoder and measure the actual frame-decoding times. These decoding times are added to the compressed video sequence. This method is proposed in [35], but requires a reference decoder for each target platform. Using a generic model of the frame decoding complexity eliminates this drawback. Such a complexity model for MPEG4 (seven parameters) is presented in [36]. They report accurate results (error $\leq$ 5%). The drawback of their method, however, is the necessity to modify MPEG4 encoders to include the complexity parameters in each frame header.

To ensure backward compatibility and general acceptance, modification of video sequences (to include clock scheduling information) must be avoided whenever possible. Therefore, an interesting question is which property from the H.263 frame gives a good estimation of the AET of the frame decoding process and can be obtained without being adding *any* knowledge to the H.263 compressed bitstream. We found that the combination of frame type and frame size yields an estimation that is simple, yet accurate. A similar estimator for MPEG2 is presented in [37].

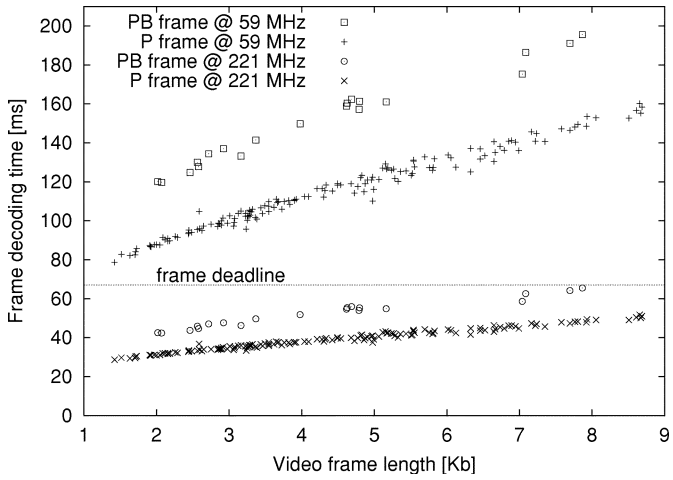


Fig. 5. Decoding time versus frame size and type.

Fig. 5 plots the decoding time versus the frame size for the Carphone test sequence on the LART platform. Two frequencies are used to decode frames of both P and PB type. The figure shows that video decoding is a demanding application: at the lowest clock frequency of our LART platform none of the frames can be decoded within the required 67 ms (i.e., 15 fps). Furthermore, running the processor on a high speed (221 MHz) is only necessary for the largest PB-frames. The cost of decoding a two image PB-frame is consistently higher than a single image P-frame. Simple P-frames decode in roughly 75 ms at the lowest clock frequency, the most complex PB-frames take almost 200 ms to decode at this speed, a significant variance. Measurements on more test sequences show that frame decoding times are independent of the content of the test sequence itself; they only depend on the type and length [38]. Note that changing the spatial or temporal resolution keeps the linear relation between frame size and decoding time, but modifies the parameters. Fortunately, such resolution changes do not occur inside normal video sequences. The characteristics of Fig. 5 will be used to estimate the minimal processing requirements for each frame.

The type of the video frame is indicated in the frame header. Unfortunately, the frame length is not part of the header; so we cannot directly determine the most suitable AET. The frame length can optionally be added to the header by using the H.263 PEI (extra insertion information) header field. This requires changing H.263 encoders to add the frame length information to the header. A solution that modifies only the decoder is preferred. By using input buffering in the decoder it would also be possible to determine the frame length before commencing with the decoding. However, this approach would increase the decoding latency, which is a severe drawback for interactive applications such as video conferencing.

C. Implementation

We extended the Telenor H.263 decoder with an AET estimator based on the observed linear relation between frame size and clock frequency for equal frame types (Fig. 5). Three modes are supported by our enhancement: *optimal*, *feed forward*, and *feed backward*. The difference between the three modes is the

knowledge about the frame decoding times. The optimal mode uses *a priori* knowledge about the decoding requirements. By using offline analysis, the exact processing requirements are determined for each video frame and made available to the decoder, similar to [35]. This mode provides a lower bound in terms of power consumption for the clock schedule. The feed-forward mode uses *a priori* knowledge about the frame length through the PEI header field. The feed backward mode does not require any modifications to the encoder or compressed bitstream, and uses intratask information updates to adjust mispredicted AETs. Our H.263 decoder communicates its processing requirements for the next deadline with the EPS scheduler once per frame in both feed forward and optimal mode and twice in feed backward mode.

The power-aware decoder in feed backward mode uses several heuristics to estimate the processing requirements as accurately as possible. Frame statistics are kept by frame type. Initially, the decoder requests the maximum processing capacity for the first few frames of a sequence, until an estimation of the time-framelength relation becomes available (using least squares fit). The best speed for the previous P-frame is used as a starting point for the current P-frame. When the upper half of the video frame is decoded an intratask update is calculated and send to the clock scheduler. The decoding time and size of the first 50% of the macroblocks is used to determine the decoding progress.

The remaining 50% of the macroblocks must also be decoded in the same frame time. When the decoder is running ahead or behind, the estimated time-framelength relation is used to calculate the new processing requirements. Unfortunately, the complexity of the upper half of the image of a video frame is not always equal to the bottom half. To compensate for this, we use the complexity ratio of the upper and lower half of the image from the previous frame to update the AET. We thereby assume that the complexity ratio is a slow-changing parameter in a video sequence.

At the start of a frame, the speed of the previous frame is only maintained if there is no frame type change. When a PB-frame follows a P-frame, the speed of the last PB-frame is used because the previous P-frame has a lower processing requirement.

The "frame_type_len" estimator from [37] also uses the type and length of the previous decoded frames to estimate the decoding time of the current frame. For their calculations they require the offline calculated relation between frame-size and decoding time. Our implementation is similar to the frame_type_len estimator, but we created an online version that uses intratask information updates.

IV. Energy Priority Scheduling

In application-directed clock scheduling, applications specify their AET to the next deadline and use intratask information updates to increase the power efficiency of the clock schedule. Our *energy priority scheduler* is an incremental online algorithm that dynamically adjusts the clock schedule when a new task enters the system or an old task completes its execution.

TABLE II
Workload Descriptions

		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		$4-6$

A. Model

This section defines a model for clock scheduling. The model combines and enhances the models presented in [16] and [39]. Each real-time task j is defined by:

- s_j starting time;
- d_j deadline time;
- e_j execution time at highest speed.

The execution 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;
- $\mathrm{run}(t)$ task that is executed on the processor at time t;

We further define the following parameters:

- $N_j(tr)$ number of tasks overlapping with time region tr besides task j;
- $N_j = \sum_{tr \subseteq [s_j, d_j]} (\|tr\|/(d_j - s_j)) N_j(tr)$ average number of other tasks besides task j;
- $f_j = e_j/d_j - s_j$ flat processor rate of task j,using the least amount of energy;
- u_j = the processor utilization currently scheduled in time region tr_j.

B. Algorithm

Before describing our algorithm, we first present two examples that motivate the scheduling heuristic we employ. Table II 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.* [16] 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 Fig. 6).

The *average rate* schedule is not optimal since A and B can be scheduled back-to-back as shown in Fig. 7. (Running at a constant speed is more energy efficient than with a varying speed).

A first improvement to the *average rate* heuristic, is to take into account the other tasks already scheduled. When scheduling a next task, we can compute the (water) level above the current schedule (contour) to fit in the computational demands (area) of the task. The *task leveling* idea is outlined in Fig. 8.

Applying task-leveling to the first example yields the optimum (Fig. 7) when scheduling task A first, followed by B. Scheduling B first and then A, however, still yields the inferior schedule shown in Fig. 6.

Our second improvement is to account for overlapping tasks that can be pushed aside. Consider the second case in Table II, which adds a third task C to the optimal schedule in Fig. 7. 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

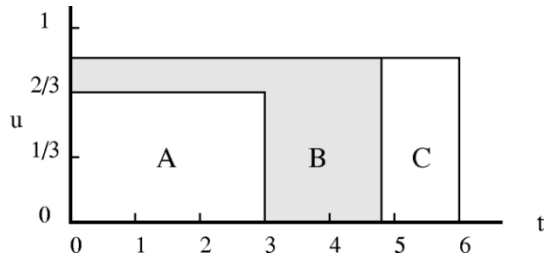


Fig. 6. *Average Rate* schedule for case 1.

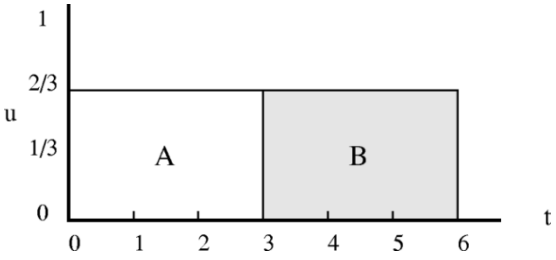


Fig. 7. Optimal schedule for case 1.

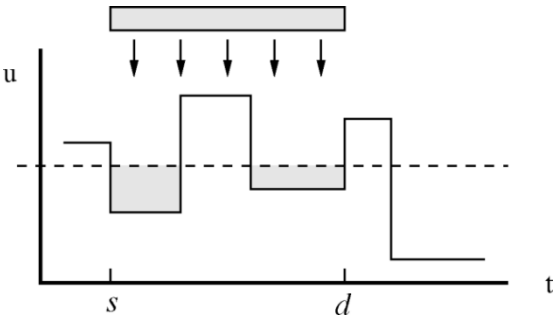


Fig. 8. Task leveling.

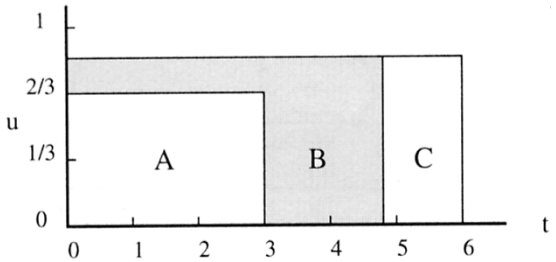


Fig. 9. Optimal schedule for case 2.

optimal schedule (an equal load of 5/6 across the entire $[0, 6]$ interval). In step 1, we determine the maximum processor utilization $u_{\max}$ on the interval $[s_C, d_C]$, which is 2/3 (cf. interval $[4, 6]$ in Fig. 7). In step 2, 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 step 3, we determine all overlapping tasks (the set T) that overlap with C; T equals $\{B\}$. In step 4, we compute the water level (5/6) above the contour of $T + C$ that accommodates the remainder of C. Finally, we reschedule tasks from the set T to create space in the interval $[s_C, d_C]$; see Fig. 9.

Rescheduling in the final step is not always possible due to deadlines regarding tasks T, in which case steps 4 and 5 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 zero) 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.

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 $|T|$ 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, $tr_l \subseteq [s_t, d_t]$, $\text{util}(t, tr_i) > 0,\ t \neq j$

3.2.3 Determine spill intervals $tr_k \in \{tr_l\} \setminus \{tr_i\}$, $u_k = u_i$

3.2.4 Define

u_{up} = lowest processor utilization on $\{tr_l\} \setminus \{tr_k\}$ (or 1 if $\{tr_l\} \setminus \{tr_k\} = \emptyset$)

$$L_i = \sum \|tr_i\|$$

$$L_k = \sum \|tr_k\|$$

$$\delta = \begin{cases} \min\left(u_{up} - u_i,\ \dfrac{\text{remain}(e_j)}{L_i}\right) \\ \quad \text{if } L_k = 0 \\ \min\left(\dfrac{L_i u_i}{L_k},\ \max\left(u_{up},\ \dfrac{\text{remain}(e_j)}{L_i}\right) - u_i\right) \\ \quad \text{otherwise} \end{cases}$$

3.2.5 Set processor utilizations u_l to $u_i + \delta$ and reschedule tasks (including j) on tr_l and tr_i accordingly.

4 Regroup tasks spread across multiple intervals.

The details of our energy priority scheduler are presented in Algorithm 1. step 2 calculates the priorities of the tasks. For example, in case 2 above the priorities are set to $p_A = (2/3)((2/3) \times 1)$, $p_B = (5/18)((2/6) \times (5/6))$, and

$p_C = (1/2)((1/2) \times 1)$. Therefore, EPS will schedule task B first, then C, and finally A. Note that this order is independent of the actual task arrivals, which avoids the sensitivity observed for the simpler heuristics discussed above. In step 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 spill intervals are the time regions of the overlapping tasks, not including $[s_j, d_j]$, were the processor utilization is equal to u_i and the utilization will be increased to make room for task j. Note that we only consider overlapping tasks that are actually scheduled on tr_i by including the "util$(t, tr_i) > 0$" condition. If there are no spill intervals ($L_k = 0$), for example, when scheduling the first task, the remaining work of j will be scheduled on top of the tr_i time regions. Otherwise, the work of the overlapping tasks is spilled from the tr_i intervals to the tr_k intervals. The actual increase (δ) is bound by the amount of work that can be spilled ($L_i u_i$), the remainder of j that still needs to be scheduled, 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. If nonoverlapping tasks were also rescheduled in step 3.2.n, both the complexity and the ability of EPS to find the optimal schedule would increase. A heuristic such as EPS will fail to find the optimal schedule in complex workloads with many tasks. 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. Fortunately, such workloads are not common for wearable devices where users typically run one or two concurrent applications. 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 $O(n \log n)$.

The presented energy priority algorithm makes a complete new schedule each time a new task arrives. When implementing this algorithm several additions must be made such as properly updating the task list T when a new task arrives and the current running task is not yet finished. For an incremental version of the scheduling algorithm the following procedure is used: each time a new task j arrives, the set of intervals tr_i is extended, followed by one round of scheduling for task j (no looping over all tasks in step 3).

The energy priority algorithm must support sporadic tasks in a real-time OS context. The algorithm can support periodic tasks by adding a parameter w that indicates the window size for periodic task scheduling. Before step 0, every periodic tasks is converted in up to w sporadic tasks and added to the task set T. The periodic task with the shortest repetition period $r_{\min}$ bounds the interval $[0, r_{\min} w]$ in which the periodic tasks are converted into multiple sporadic tasks. For example, T is extended with A ($r_a = 5$, $e_a = 2$) and B ($r_b = 9$, $e_b = 3$). When $w = 10$ the interval $[0, 50]$ is scheduled with ten sporadic tasks A and six sporadic tasks B.

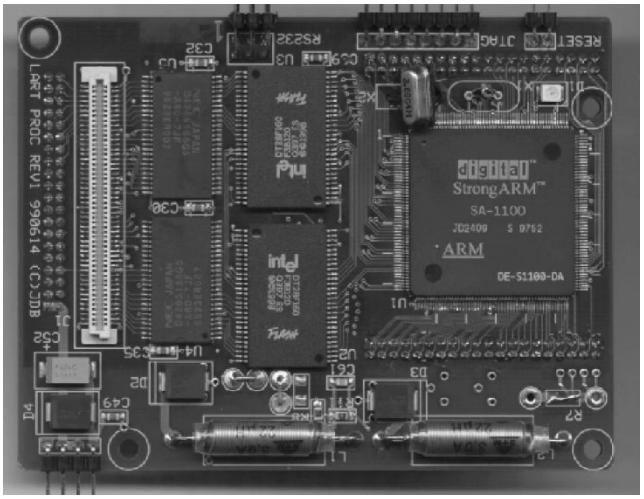


Fig. 10. Low-power StrongARM embedded Linux platform (LART).

V. Results

To demonstrate the effectiveness of application-directed clock scheduling we have performed power measurements on a complete system consisting of variable-voltage hardware, OS driver, clock scheduling daemon and algorithm, and power-aware video decoder.

A. Experimental Platform

The embedded StrongARM processor board displayed in Fig. 10 forms the heart of the wearable augmented-reality terminal that we are developing within the UbiCom project [40]. The board, named LART, has a size of 10 $\times$ 7.5 cm, a weight of 50 g, 32 MB of volatile memory, 4 MB of nonvolatile memory, a SA-1100 190 MHz processor, and various I/O capabilities. The LART has a programmable voltage regulator to control the voltage of the processor core. In Section I-A, we already discussed the relation between processor frequency (59–236 MHz, steps of 14.7 MHz) and core voltage (0.79–1.5 V), see Fig. 1.

The LART runs under control of the Linux operating system (Version 2.4.0), which has been enhanced to support frequency and voltage scaling. We added a kernel module that reads the required frequency from `/proc`, a Linux pseudo-filesystem used as a generic interface to kernel data structures, changes the clock frequency, and adjust the core voltage. It subsequently recalibrates the kernel's internal delay routines, in particular those that busy-wait by counting instruction cycles. In addition, the kernel module adjusts the memory parameters that control the timings of the read/write cycles on the external bus. The code has been structured such that it may be interrupted and does not depend on external memory, which is temporarily unavailable

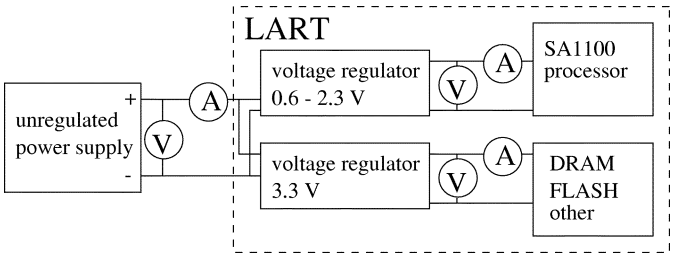


Fig. 11. Measurement setup.

during a clock frequency change. The SA-1100 is not designed for on the fly clock frequency changes. All DMA transfers are interrupted during a change, causing problems for the DMA transfers of LCD video data. The LCD device driver needs to be informed of frequency changes and temporarily halt the DMA transfer. All LART design schematics and kernel modules are publically available [41].

To measure the power consumption of the LART, we used the configuration shown in Fig. 11. The unregulated power of a battery is converted into a fixed 3.3 V for all the components on the board, except the processor core (CPU + cache) which is supplied by a variable voltage regulator. The fixed/variable voltage and current are sampled using a small sense resistor at a rate of 2.5 kHz. The standard deviation of the measurements is within 2% of the mean.

We implemented a clock scheduler that mediates between applications and the basic OS driver controlling the core voltage and processor speed. 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 (AET and deadline). We use a combination of interval-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 processing needs (cycle count, minimum speed, or AET). Before running the 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. The EPS algorithm may preempt tasks. PowerScale uses the Linux process scheduler for this purpose and sends STOP and CONT signals to processes that must be preempted and resumed, respectively.

The interval-based component of PowerScale serves two purposes 1) it supports traditional applications and 2) it corrects for miss predicted workloads by power-aware applications. Traditional applications do not register their workload with EPS, hence, the speeds determined by EPS will be too low. Mispredicted workloads can cause EPS to determine a too low as well as too high speed. The interval scheduler within PowerScale monitors the Linux process scheduler statistics to see if the EPS schedule needs to be adjusted or not. 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. In effect the interval scheduler provides negative feedback to the speed schedule produced by EPS. To ensure stability the interval scheduler uses relatively long intervals in which EPS can issue multiple speed changes. Therefore, the speed correction is applied as a delta (e.g., two steps up) to the rapidly changing EPS schedule.

The interval scheduler itself is quite flexible and operates with a parametrized interval length (a multiple of 10 ms, the granularity of Linux's 100-Hz internal timer). This allows it to operate stand alone (i.e., without EPS); a short interval length should be used to be able to closely follow the changes in the workload. To further improve the responsiveness of the system we employ the following heuristic. On consecutive speed increments we double the correction factor (exponential increase). On consecutive speed decrements, however, we simply step down to the next lower correction (linear decrease) since running at a too high speed does not impact responsiveness; only energy is wasted. The correction factor (delta) is applied to a fixed maximum performance schedule (236 MHz).

A modification of the power-aware video decoder was required to work around the poor granularity of the internal Linux timer (100 Hz). 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 due to the poor Linux timer granularity, 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 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.

B. Video Decoder Modes

We used the experimental setup discussed in Section V-A to measure the power consumption of our extended H.263 decoder (Section III) on top of PowerScale.

Table III shows the average power consumption of the LART platform for decoding a test sequence for the three supported modes of the decoder: feed backward (FB), feed forward (FF), and optimal (opt). For comparison the "236" column shows the average power consumption with clock scheduling disabled and using a fixed clock frequency of 236 MHz. The average power is computed by measuring the total energy consumed by the LART and dividing that by the duration of the test sequence. The test sequences are stored in the RAM-disk provided by Linux, hence, little energy is needed to retrieve them.

The measurements show that the FB mode reduces energy consumption considerably compared with running at 236 MHz,

TABLE III
AVERAGE POWER CONSUMPTION [mW]

Sequence	total (core + fixed)				fixed
	236	FB	FF	opt	
Grandma	404.6	244.5	243.1	242.2	209.6
Salesman	417.0	262.7	254.2	245.5	208.3
Trevor	496.1	362.8	355.6	347.9	249.4
Carphone	556.5	368.7	357.4	351.2	263.5
Foreman	571.6	389.7	380.3	374.7	283.5

for example, the average power dissipated when decoding the Grandma sequence drops from 404.6 to 244.5 mW. The reduction for FB ranges from 1.37 (Trevor) to 1.66 (Grandma). Providing the decoder with additional information (FF and optimal) does indeed reduce energy consumption further, but the gain is limited. In the case of FF, the reduction ranges from 1.40 (Trevor) to 1.67 (Grandma). The optimal policy achieves reductions in the range of 1.43 (Trevor) to 1.70 (Salesman).

The differences between the various policies is small because voltage scaling only reduces the power consumption of the processor core. The last column in Table III presents the power consumed by the components (memory, bus, etc.) supplied from the fixed 3.3 V. It shows that the fraction of the total power that can be attributed to nonCPU subsystems is considerable. For example, when decoding the Grandma sequence at 236 MHz, 209.6 out of 404.6 mW are consumed by nonCPU subsystems. The fixed fraction of the total power consumption ranges from 47.4% (Carphone) to 51.8% (Grandma). As a consequence, the maximum power reduction that can be obtained by controlling the processor speed and core voltage is limited to roughly a factor of two. We expect, however, that this limit can be increased by optimizing the H.263 decoder to take the size of the cache, which is part of the scalable processor core, into account to reduce the memory traffic. For example, large look-up tables are ineffective on the LART with its small data cache of 8 kB, and degrade performance.

When considering only the power consumed by the processor core, FB achieves a significant reduction of 2.25 (Trevor) to 6.45 (Grandma). The reduction by FF ranges from 2.33 (Trevor) to 6.63 (Grandma), and the optimal policy results in a reduction of 2.59 (Trevor) to 7.02 (Salesman). The relatively small difference between the FB, FF, and optimal mode indicates that *a priori* knowledge of frame length (FF) or complete processing requirements (opt) provides only a small benefit. Thus, standard H.263 video sequences can be decoded efficiently (power wise) using the feedback mode.

C. Application-Directed Versus OS Scheduling

To show the advantage of application-directed clock scheduling over interval scheduling, we study the behavior of the bursty video-decoding application in detail. We use the carphone test sequence since subsequent frames in this video often differ considerably in size and sometimes in type (see Fig. 4). Decoding a frame at a too high speed results in wasted energy; decoding at a too-low speed results in a missed deadline. Our modified H.263 decoder reports the accumulated miss times

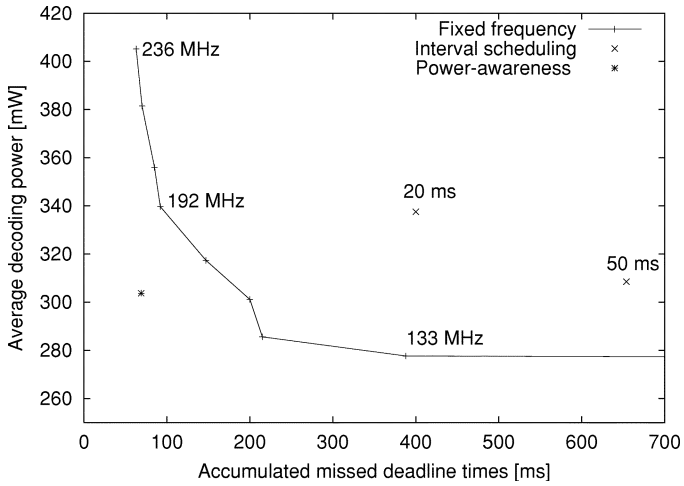


Fig. 12. Power-quality tradeoff.

at the end. With this quality measure it is possible to study the tradeoff between power and quality. Our accumulated miss times metric is similar to the clipped-delay metric in the simulations by Pering [30].

Fig. 12 shows the power-quality tradeoff for application-directed clock scheduling, interval-scheduling, and decoding at fixed speeds. Note that in all cases deadlines are missed. This is caused by the initial I-frame in the carphone sequence that cannot be decoded within 67 ms, even at the highest frequency. The solid line in Fig. 12 shows the effect of decreasing the (fixed) frequency from 236 MHz (405 mW, 63 ms) down to 133 MHz (278 mW, 388 ms). The power consumption goes down at the expense of additional deadline misses since the number of frames that cannot be decoded within 67 ms increases when the clock frequency lowers.

In interval-based mode, PowerScale performs worse than running at a fixed speed. For example, with a 20-ms interval setting PowerScale operates with an average power of 337 mW and causes 400 ms of missed deadlines; running at a fixed speed of 192 MHz requires the same power, but reduces the missed deadlines to only 92 ms, while running at 133 MHz incurs a similar miss time, but requires less power (278 mW). The problem for the interval scheduler, is that a short-time average is not a good predictor for the speed at which to decode the next frame. Increasing the interval length makes the scheduler behave more like a fixed-speed scheduler; with a 50-ms interval the power consumption gap to the fixed schedules (solid line) is smaller than at 20 ms, but many more deadlines are missed. Without additional knowledge an interval scheduler will never be able to handle bursty workloads well.

Using the AET information from the power-aware video decoder results in substantial power savings since the workload description allows PowerScale (EPS mode) to select the right decoding speed in most cases. The decoding of the carphone sequence requires only 304 mW (100 mW CPU, 204 mW non-CPU), and misses just a few deadlines: 67 ms in total, of which the largest fraction is caused by the too-demanding initial I-frame. For comparison, decoding at the fixed frequency of 236 MHz consumes 405 mW (198 mW CPU, 207 mW non-CPU) and delivers the same quality: 63 ms of accumulated

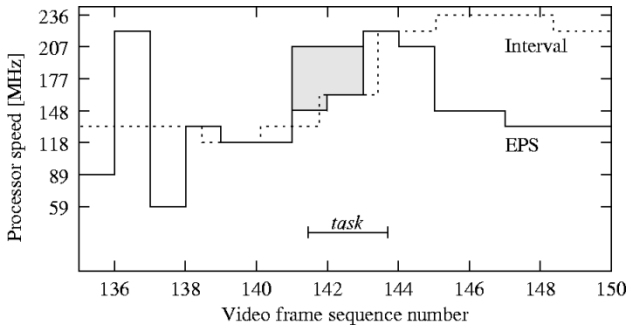


Fig. 13. Clock schedules executed by PowerScale.

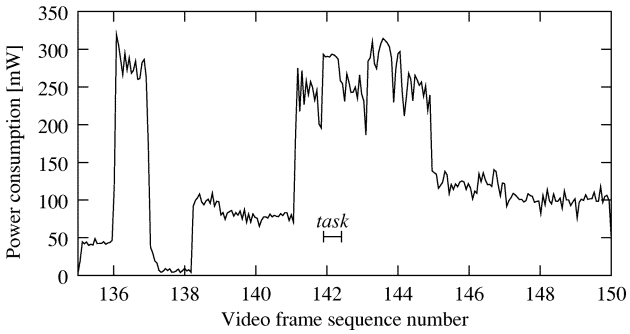


Fig. 14. Processor power consumption of EPS.

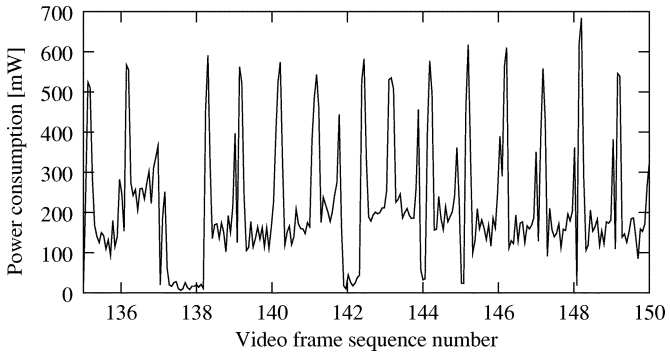


Fig. 15. NonCPU power consumption of EPS.

deadline misses. Thus, application-directed voltage scaling reduces the power consumption of the processor core with a factor of two. The total system power, however, is only reduced by 25% because of the power consumed by the nonCPU subsystems supplied by the fixed 3.3 V.

D. Multiple Applications

We now demonstrate the ability of the EPS algorithm to combine the processing needs of multiple applications and create a power-efficient clock schedule. In the following experiment, the carphone sequence is decoded in conjunction with a synthetic application. The synthetic application is set to execute for a short period (150 ms, 40 MHz) near the end of the video sequence (frames 141–143). Both the video decoder and the synthetic task register their processing requirements (AETs and cycle count, respectively) with the PowerScale scheduler. We log the speed changes initiated by PowerScale during the experiment, and measure the power consumption of the processor core. The solid line in Fig. 13 shows the actions of the PowerScale scheduler 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 shaded area shows the impact of the synthetic task on the EPS schedule: the speed is raised to 207 MHz. For comparison the dotted line in Fig. 13 shows the behavior of PowerScale when running in interval-based mode. The resulting speed is either too low (e.g., frame 136) or too high (e.g., frames 144–150).

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 Fig. 13. The synthetic task thus overlaps with frames 141, 142, and 143. 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), and 143 (207 MHz). The final schedule raises the processor speed during the decoding of frames 141 and 142 (i.e., the shaded area in Fig. 13). 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 (Fig. 14) shows a shape that is quite similar to the clock schedule executed by PowerScale in EPS-mode (Fig. 13). 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 (frame 136 : 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 Fig. 14. 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.

Fig. 14 shows the power dissipated by the CPU only. Fig. 15 shows the power dissipation of the 3.3 V part of our system (memory, bus, etc.). Despite the clock speed changes induced by PowerScale, the system load in Fig. 15 shows a quite regular pattern, except form frames 136, 137, and 142. The decoding of frames 136 and 137 involves a PB sequence where all computation is performed in the first frame (136), hence, the "zero" power consumption in frame 137. The drop to zero in frame 142 is caused by the execution of the synthetic task that does not reference any memory, but only exercises the CPU. The high peaks at the beginning of each frame are caused by the video decoder performing the runlength decoding of the compressed frame. This involves fetching data from the RAM disk, where the carphone sequence is stored, into the cache over the external bus. After this burst of memory traffic the decoder starts processing the data, which requires more computation, and the power dissipation of the memory subsystem drops to a level around 200 mW. Note that the average nonCPU power (202 mW) exceeds the average processor power (118 mW), which limits the overall effectiveness of voltage scaling.

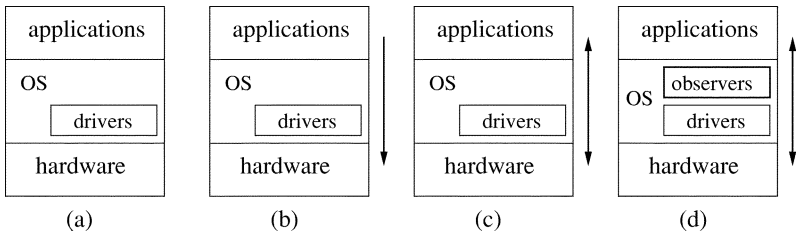


Fig. 16. Four power management frameworks.

VI. Conclusions and Future Work

Clock (and voltage) scheduling is an important technique to reduce the 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. Fully utilizing these gains, however, has proven to be difficult.

Various researchers have addressed the underlying clock scheduling problem. We classified the different approaches based on the quality and availability of information about the (future) workload, since that mainly determines the energy efficiency of the resulting clock schedules. The application-directed approach has the best information on the workload, and can create an energy-efficient clock schedule that meets the demands of the applications while minimizing the required energy.

Application-directed clock scheduling in a general-purpose OS context requires applications to become power-aware and explicitly specify their processing requirements and deadlines. In contrast to real-time systems, applications should specify their AET instead of the conservative WCET. This results in lower power consumption, since the AET is a better predictor for the true execution time than WCET. Another advantage is that an application may update its processing requirement when its demand (AET) changes. In the case of video decoding, we have shown that adding power-awareness can be done effectively. Our AET estimator is both accurate and requires no *a priori* information.

This paper describes our EPS algorithm that combines the various task requirements (AET + deadline) and yields a clock schedule that is both energy efficient and meets task deadlines. 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. EPS does not always yield the optimal schedule, but has low complexity and can be used as an incremental online algorithm.

To demonstrate the effectiveness of application-directed clock scheduling we have actually built a complete system consisting of variable-voltage hardware (StrongARM based), OS support (Linux driver), clock scheduling daemon (PowerScale), clock scheduling algorithm (EPS), and power-aware application (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 (236 MHz). This is a significant improvement over interval scheduling achieving 33% reduction. EPS achieves its reduction without missing deadlines, unlike interval scheduling that does miss deadlines. The processor only consumes a portion of the total system power. When compared with running at full speed, EPS reduces the system power with 25%.

Our future plans are to extend the application-directed method for controlling voltage scaling to power management in general. Within the UbiCom program we are developing a framework that is based on the explicit exchange of performance and power consumption information between hardware devices (CPU, hard disk, wireless link, etc.), OS, and applications. The explicit exchange of information will allow us to perform intelligent and efficient power management for the complete wearable Ubicom system.

Fig. 16 shows four power management frameworks with three different layers: hardware, OS, and application. Fig. 16(a) is the traditional framework without performance-power consumption exchange, the situation for interval-based clock schedulers. In Fig. 16(b), applications specify their future requirements to the lower layer and hardware devices can be scheduled more efficiently, as shown in this paper. Fig. 16(c) demonstrates interaction between applications and hardware. An example of such interaction would be a power-aware video decoder that meets almost all frame decoding deadlines, yet misses the deadlines for the complex and power expensive frames. Such a decoder would extend the single power-awareness data point in Fig. 12 into a curve that is more power efficient than the fixed frequency curve. Fig. 16(d) adds observers that log all application requests and try to predict future requests for applications that do not specify their hardware needs (similar to integrated schedulers). The purpose of including observers is to improve the energy efficiency of legacy codes.

Currently, we are working on an implementation of framework (d) [Fig. 16(d)] on our LART platform. The target application is wireless audio and video playback with a guaranteed battery lifetime that is specified by the user. Using the power consumption information of the hardware devices (CPU, hard disk, wireless link) and the application's ability to scale the image and sound quality, we can infer the control settings that provide the best quality without draining the battery completely before the user-defined target time.

ACKNOWLEDGMENT

The authors would like to thank J. D. Bakker and E.Mouw for providing them with an excellent low-power platform and assisting with the measurements and interpretation. They also thank H. van Dijk for commenting on draft versions of this paper, and the three anonymous reviewers that provided them with detailed comments enhancing the readability of the final paper.

REFERENCES

[1] J. Lorch, "The complete picture of the energy consumption of a portable computer," M.S. thesis, Univ. California Berkeley, Dec. 1995.

[2] J. Lorch and A. Smith, "Scheduling techniques for reducing processor energy use in MacOS," *Wireless Networks*, 1997.

[3] T. Burd and R. Brodersen, "Processor design for portable systems," *J. VLSI Signal Processing*, Aug. 1996.

[4] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int. Symp. Low-Power Electronics Design (ISPLED)*, Aug. 1998.

[5] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos, "Data driven signal processing: an approach for energy efficient computing," in *Proc. Int. Symp. Low Power Electronics and Design*, Aug. 1996, pp. 374–352.

[6] A. P. Chandrakasan and J. Goodman, "An energy-efficient reconfigurable public-key cryptography processor," *IEEE J. Solid-State Circuits*, vol. 36, pp. 1808–1820, Nov. 2001.

[7] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama, "Variable supply-voltage scheme for low-power high-speed CMOS digital design," *IEEE J. Solid-State Circuits*, vol. 33, pp. 454–462, Mar. 1998.

[8] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld, "Policies for dynamic clock scheduling," in *Proc. Symp. Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.

[9] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic voltage scaling for portable systems," in *Proc. Design Automation Conf.*, 2001.

[10] T. Burd, T. Pering, A. Stratakos, and R. Brodersen, "A dynamic voltage scaled microprocessor system," in *Proc. Int. IEEE Solid-State Circuits Conf.*, Feb. 2000, pp. 294–295.

[11] J.-D. Bakker, K. Langendoen, and H. Sips, "LART: Flexible, low-power building blocks for wearable computers," in *Proc. Int. Workshop Smart Appliances and Wearable Computing (IWSAWC)*, Scottsdale, AZ, Apr. 2001.

[12] Intel StrongARM SA-1100 microprocessor developer's manual. [Online]. Available: http://developer.intel.com/design/strong/manuals/278 088.htm.

[13] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic voltage scaling on a low-power microprocessor," in *Proc. 7th ACM Int. Conf. Mobile Computing and Networking (Mobicom)*, Rome, Italy, July 2001, pp. 251–259.

[14] Y.-R. Lin, C.-T. Hwang, and A. C.-H. Wu, "Scheduling techniques for variable voltage low-power design," *ACM Trans. Design Automation Electron. Syst.*, vol. 2, no. 2, pp. 81–97, Apr. 1997.

[15] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava, "Power optimization of variable voltage core-based systems," in *Proc. 36th Design Automation Conf.*, June 1998, pp. 176–181.

[16] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proc. 36th IEEE Symp. Foundations Computer Science*, Oct. 1995, pp. 374–382.

[17] C. Hsu, U. Kremer, and M. Hsiao, "Compiler-directed dynamic frequency and voltage scaling," in *Proc. Workshop Power-Aware Computer Systems*, 2000.

[18] M. G. Harmon, T. P. Baker, and D. B. Whalley, "A retargetable technique for predicting execution time," in *Proc. IEEE Real-Time Systems Symp.*, 1992, pp. 68–77.

[19] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *Design Automation Electron. Syst.*, vol. 4, no. 3, pp. 257–279, 1999.

[20] H.-H. Chu and K. Nahrstedt, "A soft real time scheduling server in UNIX operating system," in *Proc. Interactive Distributed Multimedia Systems and Telecommunication Services*, 1997, pp. 153–162.

[21] T. Pering, T. Burd, and R. Brodersen, "Voltage scheduling in the lpARM microprocessor system," in *Proc. Int. Symp. Low-Power Electronics and Design (ISPLED)*, July 2000.

[22] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1997, pp. 598–604.

[23] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. Design Automation Conf.*, 1999, pp. 134–139.

[24] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. 18th ACM Symp. Operating Systems Principles*, 2001.

[25] Transmeta-corporation. The technology behind the Crusoe processor. [Online]. Available: http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf.

[26] L. Geppert and T. S. Perry, "Transmeta's magic show," *IEEE Spectrum*, vol. 37, no. 5, pp. 26–33, May 2000.

[27] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. Operating Systems Design and Implementation (OSDI)*, Nov. 1994, pp. 13–23.

[28] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power CPU," in *Proc. Int. Conf. Mobile Computing and Networking (MobiCom)*, Berkeley, CA, Nov. 1995.

[29] Y. Lee and C. Krishna, "Voltage-clock scaling for low energy consumption in real-time embedded systems," in *Proc. 6th Int. Conf. Real-Time Computing Systems and Applications*, 1998.

[30] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. ISPLED*, Aug. 1998.

[31] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic performance-setting for dynamic voltage scaling," in *Proc. 7th ACM Int. Conf. Mobile Computing and Networking (Mobicom)*, Rome, Italy, July 2001.

[32] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," in *Proc. Sigmetrics 2001*, Cambridge, MA, June 2001.

[33] D. Shin, S. Lee, and J. Kim, "Intra-task voltage scheduling for low-energy hard real-time applications," *IEEE Design Test Comput.*, Mar. 2001.

[34] K. Rijkse, "H.263: Video coding for low-bit-rate communication," *IEEE Commun. Mag.*, vol. 34, pp. 42–45, Dec. 1996.

[35] L. O. Burchard and P. Altenbernd, "Worst-case execution times analysis of MPEG-decoding," in *Proc. 10th Euromicro Conf. Real Time Systems (WRTS)*, 1999.

[36] M. Mattavelli and S. Brunetton, "Implementing real-time video decoding on multimedia processors by complexity prediction techniques," *IEEE Trans. Consumer Electron.*, vol. 44, pp. 760–767, Aug. 1998.

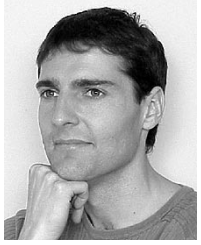
[37] A. C. Bavier, A. B. Montz, and L. L. Peterson, "Predicting MPEG execution times," in *Proc. Measurement and Modeling Computer Systems (SIGMETRICS)*, June 1998, pp. 131–140.

[38] J. Pouwelse, K. Langendoen, R. Lagendijk, and H. Sips, "Power-aware video decoding," in *Proc. 22nd Picture Coding Symp.*, Seoul, Korea, Apr. 2001.

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

[40] J. Pouwelse, K. Langendoen, and H. Sips, "A feasible low-power augmented-reality terminal," in *Proc. 2nd IEEE/ACM Int. Workshop Augmented Reality (IWAR'99)*, San Francisco, CA, Oct. 1999, pp. 55–63.

[41] J.-D. Bakker, J. A. K. Mouw, and M. A. H. G. Joosen. Linux advanced radio terminal design page. [Online]. Available: http://www.lart.tudelft.nl/.



Johan Pouwelse received the M.Sc. degree in computer science from Delft University of Technology, Delft, The Netherlands, in 1998. He is currently working toward the Ph.D. degree at the same university.

His research interests include power awareness for software, voltage scheduling, wireless link protocols, quality of service reservations, and power management APIs and has written several papers in these areas.

Koen Langendoen (M'95–A'96) received the M.Sc. degree in computer science from the Vrije Universiteit, Amsterdam, in 1988 and the Ph.D. degree in computer science from the Universiteit van Amsterdam in 1993.

He is currently an Assistant Professor with the Faculty of Information Technology and Systems, Delft University of Technology, The Netherlands. His research interests include system software for parallel processing, wearable computing, embedded systems, and wireless sensor networks.

Henk J. Sips (A'78) was born in Amsterdam, The Netherlands, on October 14, 1950. He received the M.Sc. degree in 1976 in electrical engineering and the Ph.D. degree in 1984 from Delft University of Technology, Delft, The Netherlands.

Currently, he is a Professor of Computer Science at the Delft University of Technology. His research interests include parallel systems, distributed systems, mobile systems, and low-power systems.