Power-Aware Video Decoding using Real-Time Event Handlers

Christian Poellabauer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
chris@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
schwan@cc.gatech.edu

ABSTRACT

Multimedia applications have to receive sufficient resource allocations to maintain their desired levels of Quality of Service (QoS). On the other hand, in mobile environments, the devices on which these applications must run have to minimize power consumption to prolong battery life. Our work focuses on the QoS issues in the event-driven distribution of multimedia streams between mobile users, where a source provides interactive video in the form of streams of data events to multiple remote sinks. This paper addresses the power-aware execution of event handlers at such event sinks. In particular, an adaptive approach to the dynamic selection of a suitable CPU clock frequency of a mobile device is shown superior to non-adaptive power management. This approach (a) minimizes power consumption while also (b) guaranteeing that a given event handler finishes its execution within application-specific timing constraints. This is realized by dynamically measuring the progress of event handler functions and then using this information to re-adjust the clock frequency for the current event and to select appropriate clock frequencies for future events.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Design, Performance

Keywords

energy, event service, mobile devices

1. INTRODUCTION

Mobile devices increasingly offer multimedia capabilities, thereby enabling applications like video phones, mobile teleconferencing, video on demand, and distributed multi-player games. However, a mobile device has to minimize its power consumption to prolong battery life. At the same time, to attain suitable levels of quality of service, distributed multimedia applications typically require the dynamic management of underlying computing resources, including CPU, network, disks, and displays. The challenge, therefore, is to build power-aware systems that combine two often competing requirements: (1) to minimize overall power consumption and (2) to maintain applications' desired service qualities.

Fortunately, multimedia applications can take advantage of the fact that modern mobile device hardware increasingly supports software-accessible power management mechanisms for its resources. At the CPU level, consider current StrongARM SA11xx processors, the Intel XScale 80200, the AMD Mobile K6-II+ with PowerNOW, or the Transmeta Crusoe with LongRun, all of which support the run-time selection of different frequency levels or voltage levels. They also offer the use of alternative CPU instructions, with more advanced power management support under development. At the network level, wireless cards support different transmit/receive power levels and transmission rates. And at the display level, there are different settings for brightness or backlight intensity.

As an example, consider a video conferencing application in which a mobile device receives multiple video streams that have to be displayed with certain frame rates and with small jitter to ensure sufficient video quality. This requires that the device allocates sufficient processor and network resources to the video decoding application. However, especially with wireless communications, it is likely that video frames will arrive with varying *inter-frame gaps* (e.g., bursty traffic), so that these frames have to be buffered until their display time has arrived to maintain minimal jitter.

Jitter is a well-understood problem, and frame buffering is a known solution, of course. The novel work presented in this paper focuses on a different aspect of this problem and its solution: online power management to increase the device's battery life. The idea is simple: if the device has resources beyond those needed for video decoding, why not 'slow it down' to match its CPU resources offered with those needed, assuming the device has knowledge of the desired frame rate.

The device under consideration in this paper is a Compaq iPAQ H3870 handheld with an Intel StrongARM SA1110 processor, 32MB RAM, 32MB Flash, and a PCMCIA extension sleeve carrying a Lucent Technologies Orinoco Gold wireless card with 11Mbps. The StrongARM SA1110 supports 12 clock frequencies (59Mhz to 221.2MHz in 14.7MHz steps), however, the Linux kernel used (*familiar* version 0.5.2) supports only 11 frequencies (59MHz to 206.4MHz). Dynamic frequency scaling on this device is possible for multimedia applications, with the limitation that when the clock frequency on the SA1110 is changed, the clock and all devices fed by it (LCD controller, DMA controller, serial controllers, and the OS timer) are stopped for a duration of 150μs. This leads to some implementation inefficiencies on this device, but these difficulties are likely to be reduced for future devices supporting frequency or voltage scaling, particularly for more power-optimized architectures like Intel's 80200 processors based on the XScale microarchitecture.

Motivation. Power consumption is a key issue in wireless and mobile systems, and so is Quality of Service (QoS) for multimedia applications. Our work attempts to combine adaptive QoS support for multimedia with power management, i.e., to preserve energy and to prolong battery life. This paper focuses on the receiving side of a media stream (e.g., a video decoding process), as experienced in handheld devices participating in a video conference.

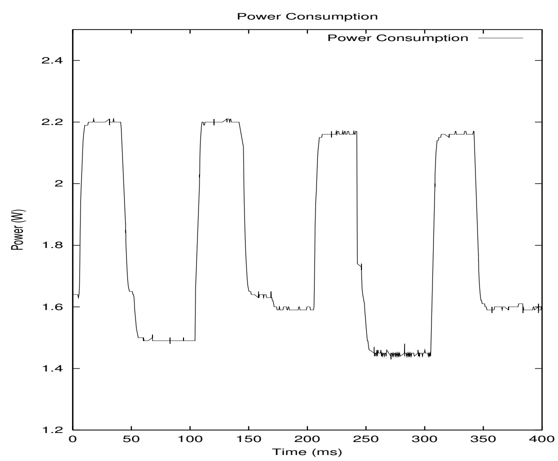

Figure 1: Video decoding on a mobile device.

Figure 1 shows a snapshot of a video decoding process on the iPAQ H3870 handheld. A video stream is received at a rate of 10 frames per second, giving the decoder 100ms for the display of each individual frame. If the device is underutilized, frame decoding can be performed faster than that, resulting in 'idle times'. It is possible to reduce such idle times by reducing the CPU clock frequency, while still meeting each frame's soft deadline for decoding [8, 15]. To enable such per-frame device power management, measurements in Figure 2 compare the energy consumption of the iPAQ with the execution time of a simple for-loop with 10^7 iterations (simulating a video decoding process) at 11 different clock frequencies. The iPAQ is run without any extension or network cards and with the LCD screen turned off. The energy consumption

$$E(Joule) = P_{active} * T_{active} + P_{idle} * T_{idle}$$

is the sum of the 'active' period of the device ($P_{active} * T_{active}$) and the 'inactive' (or idle) period of the device

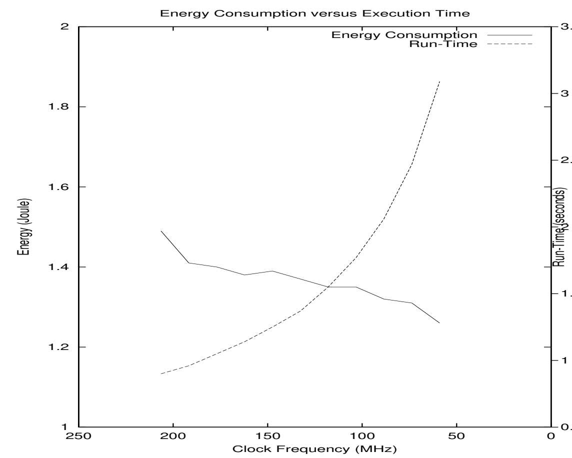

Figure 2: Energy consumption versus execution time for clock frequencies between 59MHz and 206MHz.

($P_{idle} * T_{idle}$). The energy consumption depicted in Figure 2 and in all subsequent energy graphs is computed over the period of the worst-case execution time (WCET) of the emulated function handling video frame decoding; in this experiment the handler function has a WCET of 3.09s running at 59MHz. The idle power of the iPAQ is 0.29W.

The key result depicted in Figure 2 is that although the run-time of the examined code increases by more than 2 seconds when CPU frequency is scaled from 206MHz to 59MHz, energy consumption is reduced by 200mJ (P_{active} for 206MHz: 0.92W, P_{active} for 59MHz: 0.41W). These graphs indicate that there is a possibility to save energy by 'intelligently' slowing down processing on a mobile device. You can also observe that at some frequencies the energy consumption rises. We disregard this behavior in this paper, but point out that others (e.g., in [9]) have investigated this issue.

Solution Approach. Event services or publish-subscribe mechanisms [2, 4, 7, 16] are increasingly being deployed in applications that range from remote sensing, to multimedia and video streaming [1, 10], to transactional systems [3]. We adopt this approach to enable per-frame power management for multimedia applications. That is, video or audio frames are distributed as data events from an event source – or publisher – to an event sink – or subscriber. Upon reception of an event, the event service invokes a *handler function*. For example, in a video conferencing application, the event handler can have the task of uncompressing, decoding, and displaying a video frame. *Real-time events* have deadlines associated with them, that is, the execution of the associated event handler has to terminate within defined time limits. Deadlines can be part of the received event (*explicit deadlines*), which means that they are determined by the event producer; or they can be derived from the application's desired quality of service (i.e., *implicit deadlines*), e.g., derived from the desired frame rate of a video stream.

This paper utilizes implicit event deadlines along with knowledge about the execution times of handler functions. The idea is to reduce power for event handling by choosing clock frequencies that permit handlers to execute within their deadlines while also reducing power usage. Since handler execution times cannot be assumed to be static, however, we also dynamically measure these times to attain a handler *execution history*, and we then use this history to dynamically adjust clock frequencies and to predict future run-times of handler executions, resulting in what we call *power-aware event handling*.

Contributions and Related Work. Frequency scaling [6, 9] and voltage scaling [11, 18] have been investigated in recent research. Both have been shown to be useful to reduce power consumption for a variety of application scenarios, including real-time systems [5, 11]. In [17], the authors exploit slack times to integrate fixed priority scheduling with power-awareness. The exploitation of idle times to preserve power in video decoding applications has been shown feasible in previous work [8, 15]. Our work distinguishes itself from these papers by dynamically measuring the decoding progress and re-adjusting the frequency level as required. We base the distribution of media streams on a novel publish/subscribe mechanism, where event handlers are invoked at event arrival. We modify the event service to dynamically measure the progress of the handler and allow it to select an appropriate frequency level according to experience and handler-feedback. Further, we modify the CPU scheduler in the Linux operating system to initiate re-computations of handler progress and re-adjustments of frequency levels. However, our approach works independently from the chosen CPU scheduling policy and is able to re-adjust the frequency level of a handler function while it is executed. Further, with power breakpoints, the executed code can itself provide feedback to the power management mechanism. Previous approaches [11, 19, 20] address the combination of power management with CPU scheduling, while our focus is message-oriented, that is, power management activities are triggered by message (or event) arrival.

2. POWER-AWARE REAL-TIME EVENTS

This work addresses multimedia applications that are subject to dynamic variations in underlying resources and in current user needs. Examples include networked mobile sensors cooperating to deliver remotely captured data to certain sinks in real-time, adhoc-networked PDAs operating in dynamically changing contexts, and embedded computers in mobile platforms like cars and airplanes that interact to exchange time-critical information. Applications investigated by our group are ones that (1) remotely capture certain sensor data (e.g., video capturing), (2) forward such data to sinks that need it, via a shared wireless communication medium, and (3) permit sinks to dynamically specialize sensor/media information to suit their current needs and capabilities. The element of such applications explicitly studied in this paper is a video decoding process on mobile devices, focusing on sink-side event handling on power-limited devices.

2.1 KECho Event Service

The event-based approach to distributing data between remote event sources and sinks has been used in applications such as scientific computation, virtual worlds, sensor networks, or multimedia applications. KECho [12] is an anonymous and asynchronous event service semantically similar to the CORBA Event Service [4], but implemented directly on top of a network stack within the Linux kernel. The reason for its all-kernel implementation is the use of KECho for highly dynamic functions like QoS management or resource monitoring, where we (1) need low overheads and high performance (e.g., by avoiding costly system calls) and (2) need direct and fine-grained access to all kernel resources, data structures, and devices. For instance, KECho supports the cooperation between its event service and the Linux CPU scheduler to maximize event responsiveness, such as minimizing the jitter of multimedia applications [13]. Further, to integrate kernel-based with user-level resource management, KECho also offers an interface to user-level applications (via ECalls [14]).

2.2 Event Handling and Power-Awareness

Once a distributed application has established an *event channel*, events can be submitted anonymously to all channel subscribers. A subscriber has to indicate an *event handler function* at subscription time, which will be called once for each incoming event. If there is a deadline associated with an arriving event, the event service has to ensure that the handler function terminates within this deadline. Explicit deadlines are determined by the event publisher and are piggybacked onto an event at submission time. Implicit deadlines are determined by the application at the event subscriber and are derived from the context or desired quality of service of the application. Video decoders, for example, can derive event deadlines from the desired frame replay rate: all events (assuming that an event carries exactly one video frame) have deadlines exactly $\frac{1}{framerate}$ seconds apart.

In mobile systems, we not only need to ensure the desired quality of service for each application, but have to preserve power of the overall device. *Power-awareness* in the KECho event service is achieved by prolonging handler execution such that deadlines are still met, but power consumption is minimized. Although handler execution times at different frequency levels can be measured offline, 'outside' factors such as preemptions by other tasks or events can make run-times unpredictable. To react to the dynamics in event handler execution, an adaptive approach to power management is required. Note that in this work we address *soft real-time* applications, that is, applications where real-time requirements can be softened in some situations (e.g., "90% of all deadlines have to be met").

KECho has been modified with the following extensions: (1) it dynamically measures the execution time of its event handlers and computes an average over recent handler executions for each available clock frequency; (2) it dynamically selects the clock frequency by comparing event deadlines with average handler run-times; (3) the CPU scheduler is modified so that it restores a task's frequency level when the task is being scheduled, thereby allowing per-task frequency levels; and (4) the handler functions and the CPU scheduler have the ability to call back into KECho to request re-consideration of the selected frequency level, thereby implementing *power breakpoints* for event handlers.

At each power breakpoint, the actual progress of a handler is compared with its predicted run-time and the clock frequency is adjusted if necessary. These adjustment are made because slowing down a handler's execution also brings the handler closer to its deadline, thereby increasing the risk of missing the deadline. Delays or preemptions of a handler function can easily result in a missed deadline in such situations. 'Internal' delays are caused by varying content of an event, perhaps due to the increased quality or size of an image. 'External' delays are caused by other tasks competing for the same resources, as exemplified by the arrival of a higher-priority event that causes its handler function to preempt the currently running handler.

We address these situations with different approaches:
(i) KECho reacts to possible delays by re-adjusting the frequency level and maintaining desired levels of *loss tolerances* (i.e., it adaptively controls the number of missed deadlines).
(ii) Call-backs (or *power breakpoints*) in the handler code are used to initiate re-consideration of the selected frequency level, and the handler can pass event-specific information (e.g., size or complexity of an image) to KECho.
(iii) Call-backs from the CPU scheduler are used to reconsider the selected frequency (as in (ii)), but without the ability to pass event-specific information along with the call-back.

The following sections describe in detail the modifications to the KECho event service. All measurements are performed with a PicoTech dual-channel PC oscilloscope (sampling rate 100kS/s and 12 bit resolution). The batteries in both the iPAQ and the extension sleeve are disconnected, which means that the only power source is the DC adapter. A 1Ω resistor is inserted into the ground line between adapter and iPAQ to allow for the measurement of the current and therefore the power of the iPAQ. A Linux-based x86 desktop with a dual-Pentium III (each 800 MHz) and 1GB RAM is used to 'feed' video frames to the iPAQ. The kernels on both the desktop and the iPAQ are a modified Linux 2.4.17 kernel ('familiar 0.5.2' on the iPAQ and 'Redhat 7.3' on the desktop).

3. DYNAMIC CLOCK SPEED SELECTION

The KECho event service has been modified to be able to measure handler execution times and to store averages for each 'handler-frequency level' (more conservative approaches could prefer to store WCETs instead). These averages are used to select the appropriate frequency level for future events. The tables in Figure 3 store the average ex-

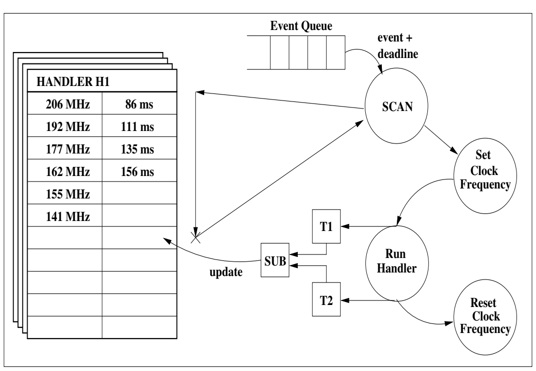

Figure 3: Clock frequency selection.

ecution times for each individual frequency level and each handler function. When an event is taken from the event queue, its associated deadline is used to find the appropriate entry in the handler-specific table. The clock frequency is set to its new value and then the handler function is executed. Before and after handler execution, timestamps (T1 and T2 in Figure 3) are obtained and used to update the average execution time in the table for the used clock frequency. Finally, after the handler function has terminated, the clock frequency is reset to its original value.

The following pseudo-code shows the frequency level selection described above. In lines 4-7, we compare the average run-time of the handler function at different frequency levels until we find either (a) the largest run-time smaller than the deadline or (b) a '0' entry. A '0' entry means that no run-times at this level have been measured yet. After the appropriate level has been found, the time is measured and the handler invoked (lines 9 and 10). Note that the default 'level' is 10 corresponding to 206.4MHz and level 0 corresponds to the slowest clock frequency of 59MHz.

```
 1:  loop {
 2:    if new_event_available {
 3:      level = 10;
 4:      while avg_runtime(level) < deadline &&
 5:             level >= 0 &&
 6:             avg_runtime(level) != 0
 7:        level--;
 8:      if avg_runtime(level) > deadline level++;
 9:      t1 = get_timestamp;
10:      invoke_handler;
11:      t2 = get_timestamp;
12:      update_avg_runtime(level, t2-t1);
13:    }
14:  }
```

After the handler terminates, we again measure the time (line 11) and then update the stored average run-time of this frequency level (line 12). In our current implementation, the average is computed using the 5 most recent handler executions. Note that the average run-time is the time from handler invocation until handler termination, i.e, including all preemptions. Using this scheme, we exploit the

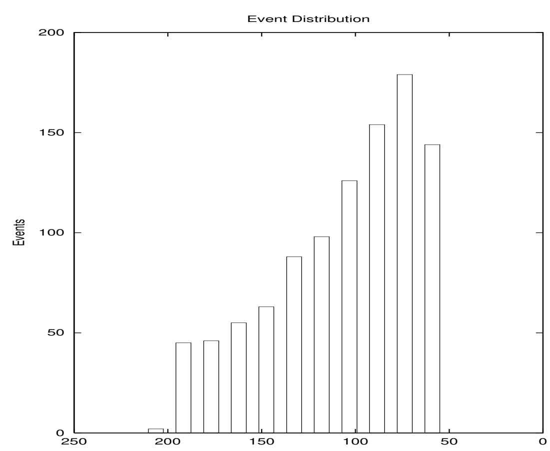

Figure 4: Event distribution.

fact that reduced frequency levels result in extended handler run-times, while also reducing power usage. For instance, Figure 4 shows the event distribution using this mechanism: 1000 events are being received and a simple handler function (in the kernel) performing a for-loop 10 million times is invoked for each event. The event deadline is randomly chosen between 20 and 60ms. Without dynamic clock selection, all handler invocations would be executed at the highest frequency level (206.4MHz). With dynamic clock selection, however, almost all handler invocations are run at lower frequency levels, with the average being 107MHz.

Figure 5 shows the power consumed by the handheld de-

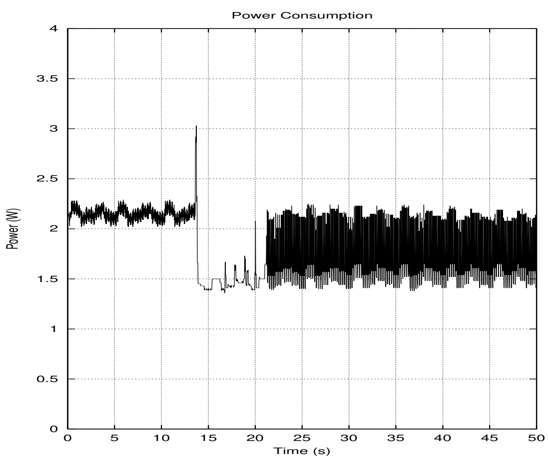

Figure 5: Power consumption without adaptive frequency selection.

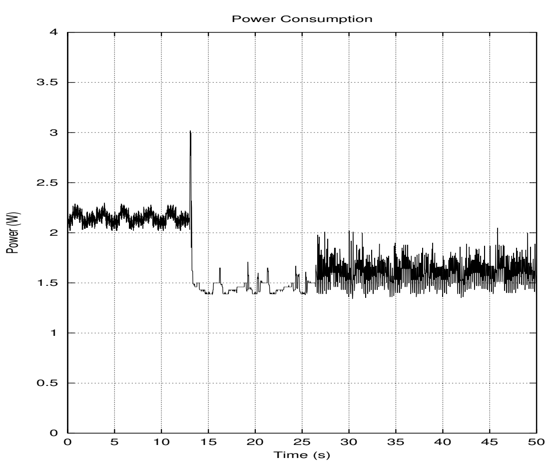

Figure 6: Power consumption with adaptive frequency selection.

vice during execution of this handler function. Events are being transmitted with a rate of 10 events per second. The first 14s show the power when the iPAQ is idle. The peak at 14s occurs when we start the KECho event service, after which the power drops to approximately 1.5W since we turn off the LCD display (to achieve higher accuracy in our measurements). For the next 7 seconds, event channel creation, subscription, etc., take place, until after 21s, the device starts receiving events. The power fluctuates between 1.4W and 2.25W, with an average of 1.8W. Figure 6 shows the same situation again, this time with adaptive power level selection. Here, the power fluctuates between 1.4W and 2W, with an average of 1.6W. Note that the channel creation time varies between both graphs, which is caused by varying response delays from a user-level *session manager* on a remote host. The session manager is contacted by each subscriber once to store or obtain a channel identifier.

4. RUN-TIME VARIATIONS

Although we dynamically measure and update the average run-time length of a handler function, variations in run-time can be large due to **external** and **internal** influences. External influences are outside of the control of the KECho event service, e.g., handler functions can be delayed or preempted by other tasks with higher priorities. Internal invariants are under the control of KECho and are caused by variations in event size or complexity (e.g., changing size or resolution of images). The handling of such variations is the topic of the following sections.

4.1 Conservative Clock Speed Selection

A first approach to prevent missed deadlines caused by internal and external delays is to choose a more conservative approach in clock speed selection. A modification to the frequency selection algorithm is to compare average handler run-times with a fraction of the deadline. The following pseudo code shows this modification, where OFFSET is a real number in the range]0..1]:

```
      while avg_runtime(level) < OFFSET * deadline &&
             level >= 0 &&
             avg_runtime(level) != 0
        level--;
```

The smaller the value of OFFSET, the smaller the probability of missed deadlines. The selection of OFFSET can by made dynamically. For example, soft real-time data streams (such as video streams) often can tolerate a limited number of missed or late packets. If a user wants a video stream with a *loss tolerance* of 10%, KECho can keep track of the number of missed deadlines and can adjust OFFSET accordingly (i.e., decrease OFFSET if the number of missed deadlines is larger than the allowable loss tolerance). Figure 7 shows

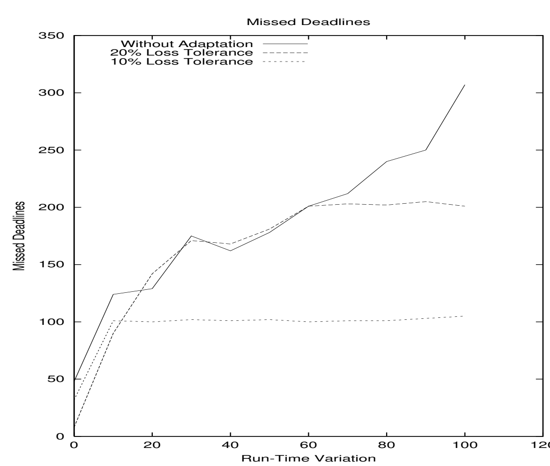

Figure 7: Missed deadlines.

the number of missed deadlines over run-time variations (in the range of 0 to 100% of the original run-time) for the handler function used in Section 1 for three different scenarios:

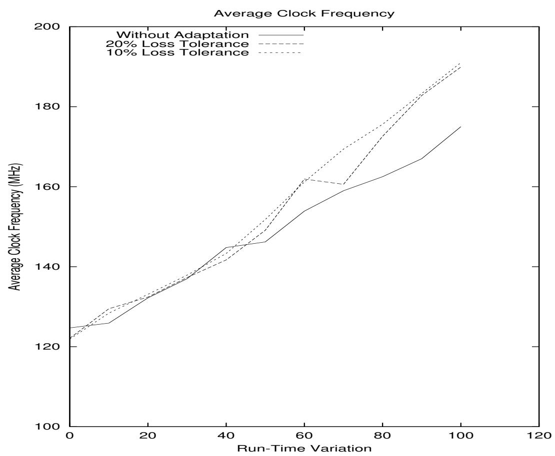

Figure 8: Average clock frequencies.

(i) OFFSET is fixed at 1.0, (ii) OFFSET is adjusted dynamically such that a loss tolerance of 20% is achieved, and (iii) OFFSET is adjusted dynamically such that a loss tolerance of 10% is achieved. It can be seen that the dynamic adjustment of the OFFSET value for loss-tolerant streams succeeds in limiting the number of missed deadlines to the desired values. In Figure 8 we compare the average clock frequencies for these three scenarios. The maximum deviation for the 'loss-tolerant' cases (10% and 20%) from the original case is 15MHz in this example.

4.2 Power Breakpoints

With *power breakpoints*, clock frequency adjustments can be made while a handler executes, that is, we introduce adaptivity at a finer granularity. Two types of breakpoints are supported in our implementation: (i) handler breakpoints and (ii) scheduler breakpoints.

Handler Breakpoints. Handler breakpoints (Figure 9) are placed directly in the handler function by the handler developer. Each time a breakpoint is reached, the handler execution is interrupted and a call-back into KECho is performed. KECho then compares the actual run-time of the handler with the predicted run-time and changes the clock frequency if necessary. This is useful for situations where external influences delay the execution of the handler function. The positioning of a power breakpoint has an influence on the usefulness of the breakpoint. For example, a breakpoint placed in the second half of the handler code may be more useful than a breakpoint in the first half. Further, a handler breakpoint has one argument which is a simple integer in the range from 0 to 10. This argument allows the handler function to inform KECho about the complexity of the currently handled event, which allows then KECho to choose a more conservative frequency level if required. As example, a video decoding handler function can 'inform' KECho about an image size or quality that deviates from the size or quality of previous images. KECho determines the clock frequency level as described before, however, it takes the breakpoint argument into consideration. For example, an argument of '2' indicates that KECho should choose a clock frequency of at least 2 levels higher than it would choose without this argument. This allows the handler to force KECho to be more conservative in frequency scaling. Figure 10 compares the number of missed deadlines with different breakpoints. The first bar indicates that 215 events out of 1000 miss their deadlines in this experiment. Note that

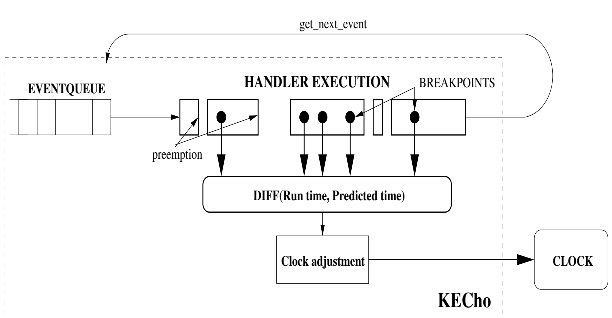

Figure 9: Handler breakpoints.

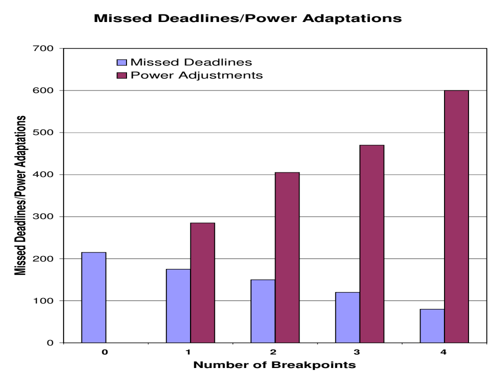

Figure 10: Missed deadlines/number of frequency adaptations.

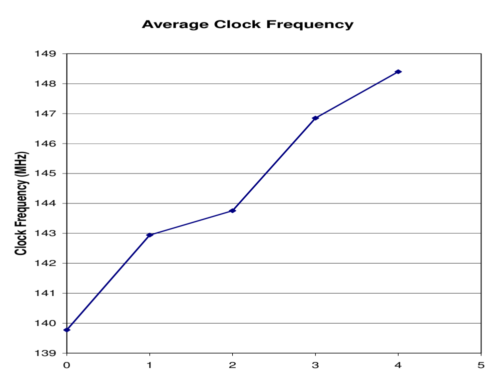

Figure 11: Average clock frequencies.

some of these events have such early deadlines that they are not able to meet them even running at the highest possible clock frequency (in this experiment about 60-70 of all events fall into this category). The following bars then indicate the missed deadlines for 1,2,3, and 4 breakpoints, which are set at equal intervals in the handler code. It can be seen that the number of missed deadlines drops from 215 with no breakpoints to 80 with 4 breakpoints. The second set of bars show the number of power adjustments that were necessary, the more breakpoints the more adjustments are being made. Figure 11 shows that the average clock frequency increases with the number of breakpoints, however, the change here is only about 10MHz between 0 and 4 breakpoints. Note that in our implementation, only if a handler finished without clock frequency adjustment during run-time, the measured run-time will be used for re-computation of the run-time average.

Scheduler Breakpoints. A second approach to the breakpoint solution relieves the handler developer from finding appropriate places in the code for the breakpoint placement. Instead, we modified the Linux CPU scheduler such that each time the scheduler is about to schedule a KECho handler, it first calls back into KECho. The advantage here is that the breakpoints are set 'automatically' by the scheduler. However, the handler is not able to inform KECho about increased complexity of an event. Further, with scheduler breakpoints the clock frequency is only re-considered when the scheduler runs and when the handler function actually is being preempted. Figure 12 shows this scenario. Each time

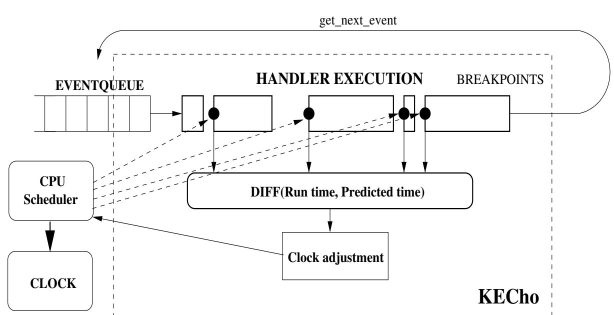

Figure 12: Scheduler breakpoints.

the CPU scheduler selects the event handler as the next task to run (after it got previously preempted by another task), the scheduler calls back into KECho, which then re-adjusts the clock frequency if necessary. The call-back functionality is independent from the scheduling policy used. To enable any CPU scheduler to make use of per-task frequency levels and to make call-backs into KECho the following changes have been applied to the Linux kernel:
(a) The task structure in linux/sched.h (**struct task_struct**) has been extended with two new entries: **clock_frequency**, which is used to store the most recently used clock frequency in kHz for this task and **kecho_task**, which is a flag that indicates if a task is a KECho event handler.
(b) After the CPU scheduler selected the next task to be run it first inspects the **kecho_task** variable in the task structure of this task and makes a call back into KECho if the entry indicates that the task is an event handler. This gives KECho the opportunity to re-compute the required clock frequency.
(c) Finally, the CPU scheduler checks the **clock_frequency** entry (which might have been modified by KECho in the previous step) and re-adjusts the clock speed if the entry differs from the current value of the clock speed.

In the case of scheduler breakpoints, the actual number of breakpoints can vary and depends on the run-time of the handler function, the number of preemptions, and the frequency of scheduler invocations.

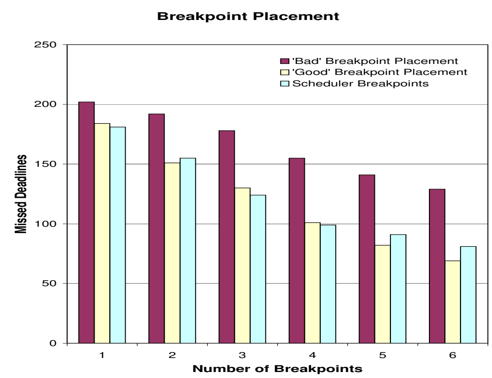

Figure 13: Comparison of 'bad', 'good', and scheduler breakpoint placement.

While in the case of scheduler breakpoints the breakpoint placement is 'performed' by the CPU scheduler, with handler breakpoints the handler developer has to identify appropriate places in the code for such call-backs. Figure 13 compares the effect of 'bad' breakpoint placement (e.g., situations where breakpoints are placed only in the first half of a handler function) with 'good' breakpoint placement (e.g., situations where the breakpoints are evenly distributed in the handler function). The latter case shows greater reductions in missed deadlines with larger numbers of breakpoints. In addition, we compare the scheduler breakpoint approach with the handler breakpoint approach: the results are comparable to the 'good' placement policy for the handler breakpoints.

5. CONCLUSIONS

Mobile devices have to coordinate the resource utilization of their applications with power management such that both are satisfied: all applications receive their desired QoS, and battery life is prolonged. This paper presents enhancements to an event-based media delivery service to support power-awareness: (1) the event service dynamically measures the execution time of its event handlers and computes an average over recent handler executions for each frequency level; (2) the event service is able to dynamically change clock frequency and does so by comparing event deadlines with average handler run-times; (3) the CPU scheduler is modified so that it restores a task's frequency level when the task is being scheduled; and (4) the handler functions and the CPU scheduler have the ability to call back into the event service to request re-consideration of the selected frequency level.

These modifications exploit idle times of the mobile device to preserve power and do this in a manner that dynamically reacts to variations in the run-time (or progress) of handler functions. Our results show (1) power can be saved by selecting slower clock frequencies depending on event deadlines, and (2) that real-time behavior can be improved by dynamically re-adjusting the chosen clock frequency during handler execution.

6. FUTURE WORK

Future work will focus on the effects of multi-dimensional resource management (several resources have to be managed cooperatively) on power management. Further, filtering is an important feature of event services. That is, processing of an event at the sending side can reduce network overheads and also power consumption of the network devices. As shown before, the iPAQ consumes about 1.5W (0.29W idle) for processing, whereas wireless cards such as the Orinoco Gold require 0.9W/1.4W for message reception/transmission. We will investigate the trade-off between added processing overhead and reduced network overhead, and we will implement an adaptive scheme to dynamically select appropriate filtering functions to minimize power consumption.

Although frequency or clock scaling can reduce power consumption, as shown in this paper, voltage scaling is preferable because of its greater impact on power reduction. By reducing frequency, the power consumption reduces linearly to the frequency. However, by reducing voltage, the power consumption reduces quadratic (i.e., $P \sim (V^2, f)$). Therefore, our future work will use newer architectures, such as the Intel XScale processor (which will be used in HP's iPAQs H3950 and H3970), which offers both voltage and frequency scaling.

7. REFERENCES

[1] D. Chambers, G. Lyons, and J. Duggan. Stream Enhancements for the CORBA Event Service. In *Proc. of the 9th ACM Multimedia Conference, Ottawa, Ontario, Canada*, October 2001.

[2] G. Eisenhauer, F. Bustamante, and K. Schwan. Event Services for High Performance Computing. In *Proceedings of High Performance Distributed Computing (HPDC)*, 2000.

[3] A. Gavrilovska, K. Schwan, and V. Oleson. A Practical Approach for 'Zero' Downtime in an Operational Information System. In *Proc. of 22nd Intl. Conference on Distributed Computing Systems (ICDCS-2002), Vienna, Austria*, 2002.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Object Event Service. In *Proc. of OOPSLA*, October 1997.

[5] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-Aware Scheduling under Timing Constraints for Mission-Critical Embedded Systems. In *Proc. of Design Automation Conference*, 2001.

[6] J. R. Lorch and A. J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proc. of the ACM SIGMETRICS Conference*, 2001.

[7] C. Ma and J. Bacon. COBEA: A CORBA-Based Event Architecture. In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies, Santa Fe, NM*, April 1998.

[8] M. Mesarina and Y. Turner. Reduced Energy Decoding of MPEG Streams. In *Proc. of Multimedia Computing and Networking, San Jose, CA*, 2002.

[9] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling. In *Proc. of the 16th Annual Intl. Conference on Supercomputing*, 2002.

[10] S. Mungee, N. Surendran, and D. C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Proc. of the 32nd Annual Hawaii Intl. Conference on System Sciences*, 1998.

[11] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proc. of the 18th SOSP, Canada*, 2001.

[12] C. Poellabauer, K. Schwan, G. Eisenhauer, and J. Kong. KECho - Event Communication for Distributed Kernel Services. In *Proc. of the Intl. Conference on Architecture of Computing Systems (ARCS'02), Karlsruhe, Germany*, April 2002.

[13] C. Poellabauer, K. Schwan, and R. West. Coordinated CPU and Event Scheduling for Distributed Multimedia Applications. In *Proc. of the 9th ACM Multimedia Conf., Ottawa, Canada*, October 2001.

[14] C. Poellabauer, K. Schwan, and R. West. Lightweight Kernel/User Communication for Real-Time and Multimedia Applications. In *Proc. of the 11th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, June 2001.

[15] J. Pouwelse, K. Langendoen, R. Lagendijk, and H. Sips. Power-Aware Video Decoding. In *Proc. of Picture Coding Symposium 2001, Seoul, Korea*, 2001.

[16] R. Rajkumar, M. Gagliardi, and L. Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proc. of the 1st IEEE Real-Time Technology and Applications Symposium*, May 1995.

[17] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proc. of Design Automation Conference*, 1999.

[18] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. D. Micheli. Dynamic Voltage Scaling and Power Management for Portable Systems. In *Proc. of Design Automation Conference*, 2001.

[19] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation (OSDI), Monterey, CA*, 1994.

[20] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. of IEEE Annual Foundations of Computer Science*, 1995.