Automatic Performance Setting for Dynamic Voltage Scaling

KRISZTIÁN FLAUTNER, STEVE REINHARDT and TREVOR MUDGE

Advanced Computer Architecture Lab, The University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109, USA

Abstract. The emphasis on processors that are both low power and high performance has resulted in the incorporation of dynamic voltage scaling into processor designs. This feature allows one to make fine granularity tradeoffs between power use and performance, provided there is a mechanism in the OS to control that tradeoff. In this paper, we describe a novel software approach to automatically controlling dynamic voltage scaling in order to optimize energy use. Our mechanism is implemented in the Linux kernel and requires no modification of user programs. Unlike previous automated approaches, our method works equally well with irregular and multiprogrammed workloads. Moreover, it has the ability to ensure that the quality of interactive performance is within user specified parameters. Our experiments show that as a result of our algorithm, processor energy savings of as much as 75% can be achieved with only a minimal impact on the user experience.

Keywords: dynamic voltage scaling, power management, performance-setting, interactive performance, response time

1. Introduction

The performance of microprocessors has been improving at an exponential rate and this trend is likely to continue for several years to come. However, increased performance does not come for free. One of the most important consequences of higher performance has been a dramatic increase in power consumption. While an Intel 386 processor consumes about 2 watts of energy, a Pentium 4 can use more than 55 watts. In a mobile environment, batteries have not kept pace with the increased energy requirements, which means that either application performance or battery time suffers. However, even in environments where energy storage is not an issue, energy cost and heat management may become problems [10].

There is still a need to continue to improve processor performance, since not all applications are "fast enough", but an increasing number are. A way to bridge the gap between high performance and low power is to allow the processor to run at different performance levels depending on the application's requirements. Some processors, such as the Intel XScale [2] and Transmeta Crusoe [8] allow the frequency of the processor to be reduced with proportional reduction in voltage. Slowing down frequency without voltage scaling is not useful, since the power savings is offset by an equal increase in execution time, yielding no reduction in the total amount of energy consumed. However, since energy is proportional to the square of the voltage, reducing the operating voltage can yield significant energy savings [14]. The central issue with processors whose performance can be changed is how the right level of performance can be obtained. The goal is to reduce the performance of the processor without causing an application to miss its deadlines (see figure 1). Completing a task before its deadline and then idling is less energy efficient than running the task more slowly to begin with, and meeting its deadline exactly.

Our aim is to design an algorithm that balances energy savings with the following requirements:
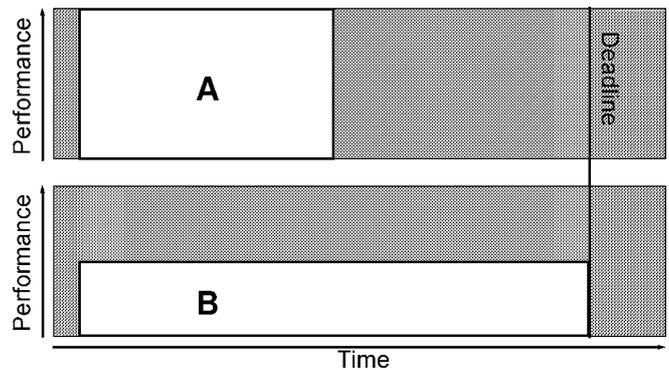
Figure 1. Performance scaling. This figure shows two different runs of the same workload. In A, the workload runs at full speed and finishes well in advance of its deadline. In B, the execution of the workload is stretched to its deadline, which allows for energy savings on processors that implement voltage scaling.

- No modification of user programs.
- Works with irregular and multiprogrammed workloads.
- Ensures that user-perceived performance does not suffer.

Previous interval-based approaches to automated performance setting did not fully achieve the goals outlined in the last two points. These approaches focus on the ratio of idle-to busy-time as the indicator of the right performance setting [5,6,13,16]. While the results looked promising for regular workloads (such as audio playback, where processor utilization is periodic), the proposed schemes do not work well for interactive or irregular applications.

The aforementioned papers point out that looking at idle time alone as the indicator of the right performance level is not sufficient. In their future work section, Weiser et al. propose an alternative approach, where jobs are classified into background, periodic and foreground classes [16]. They suggest that the added semantic information could be used to improve the scheduling algorithms. Govil et al., in their future

work section, propose a similar solution, where process type along with information specified by the processes (e.g., deadline) could be used for performance setting [5]. Our approach follows along the lines of these earlier works; however, we derive deadline and classification information automatically from the OS kernel, by examining the communication patterns between the executing tasks (section 3). This information is used to isolate execution episodes corresponding to different communication patterns. We can classify execution episodes into one of the following categories: interactive, periodic producer, and periodic consumer. These classifications can be used to derive deadlines for the execution episodes. For example, for an interactive episode, the deadline is the perception threshold, which we assume to be between 50 ms and 100 ms. The deadline for a producer is the point at which the consumer actually needs the produced data. Episode classification guides performance-setting decisions on a per-task and per-episode basis and is also used to dynamically evaluate the impact of past decisions on user-perceived performance (section 4).

We focus primarily on interactive applications, since we believe that this is one of the most difficult, but also the most important class of applications for performance scaling. We also consider the effects of a concurrently running background application (an MP3 player) and a variety of assumptions about the power and performance models on the performance-setting strategy (section 6).

2. Previous work

In the context of real-time systems, researchers have explored voltage and frequency scaling as a means of reducing power consumption. Papers [7,11,15] present algorithms and theoretical models that allow one to incorporate voltage scheduling into real-time schedulers. However, these papers are not directly applicable to general-purpose operating systems, since the workloads are expected to have well defined characteristics (periodicity, resource requirements, deadlines, etc.). Moreover, the user must explicitly specify these characteristics to the scheduler.

Our research is more closely related to the work described in [5,14,16], where performance-setting decisions are made automatically, guided by the ratio of idle time to busy time on the processor. A weakness with the existing approaches are that they are not very accurate and can be easily confused by irregular processor utilization. We improve on these algorithms by using task-level information form the OS kernel. A more detailed summary of the directly related work follows below.

The main ideas behind automatic performance setting were sketched out by Weiser et al. [16]. Their mechanism uses the amount of idle time as the guide for finding the optimum level of performance. The practical policy proposed in this paper is called PAST. In this policy the utilization for the most recent interval is computed and, if it is above a certain threshold performance, is increased, but if the interval includes mostly idle time, performance is reduced.

While the PAST algorithm looks very attractive due to its simplicity and effectiveness on some benchmarks, Govil et al. [5] point out its shortcomings and propose improvements. One of their complaints is that PAST looks back only at a single interval and thus it smooths speed poorly: the algorithm keeps on changing performance levels without coming to a steady state, missing out on opportunities to save power. To remedy the situation, the authors propose a number of algorithms that use varying amounts of prediction to improve their accuracy. They conclude that smoothing, rather than sophisticated prediction might be the most effective. Accordingly, they propose an algorithm, PEAK, that looks for recurring patterns of processor utilization, with special attention to short bursts of high utilization, and attempts to set processor speed accordingly.

Pering et al. evaluate interval-based voltage scaling algorithms for use on a handheld device [13]. One of the key contributions of this paper is the use of the clipped-delay metric, which takes into account that the length of some events can be increased without affecting the user (see section 4.1.1). In practice, the effectiveness of this technique depends on the allowed increase in delay. The algorithm may degrade performance slightly but yield significant power reduction. While the algorithms used in this paper worked, their performance fell short of optimum. Moreover, the algorithms used some specific knowledge about the executing programs, which may be impractical on a production system. Pering et al. show that while interval-based voltage scaling algorithms work well on benchmarks with regular processor utilization (such as audio playback), they do not fare well on irregular workloads, such as interactive workloads or video playback.

These conclusions are corroborated by Grunwald et al. [6]. They also find that using a weighted average of processor utilization as a guide to future utilization (the AVG_N policy) does not yield the clock speed that would maximize processor utilization. Another problem with this algorithm is that the requirement to average N intervals introduces a delay in responding to processor demand. The authors find that existing heuristics did not fare as well on an actual implementation as previous studies had suggested.

Lorch et al. improve the performance of interval-based scaling algorithms by taking a task's work requirement probability distribution into account [9]. This information is used to gradually increase the speed of the processor during execution, yielding an improvement in energy savings. A similar insight led us to the strategy for setting the performance level for interactive episodes and for deriving the *PanicThreshold* (see equation (4)).

3. Episode detection

We first applied our interactive-episode detection algorithm to measure the effects of multiprocessing on interactive performance [3]. In this section we summarize this methodology and also show how it can be extended to find periodic episodes.

The first step in determining the optimum level of performance is to distinguish the important parts of the execution from unimportant periods. We use the communication characteristics of the executing applications as the basis of this classification. Episodes are triggered by communication events with specific tasks but multiple tasks may be involved during the execution of an episode. For example, an interactive episode involving Ghostview is triggered by a message from the X server to Ghostview. In return, events are processed by the application and it may send messages to the X server, the window manager and its rendering engine (Ghostscript). All of these processes are part of the episode.

There are two principle groups of episodes: periodic and interactive. Periodic episodes may be further categorized into producer and consumer, where the communication between these episodes establishes their performance level. All other processor activity is classified as background activity. It is important to note that during its lifetime a task can fall into more than one of these classifications. For example, a music playback process may be part of an interactive episode when it is updating the GUI and be a producer when it is decoding music data.

We monitor which tasks communicate with a few well-known system tasks (such as the X server and the sound daemon). These tasks are then monitored for communication through specific system calls that are then used to classify them into one of the above categories. In addition, we collect run-time statistics about processor utilization. Thus, instead of relying on the programmer, we extract the necessary information from the system automatically, using simple changes to the OS kernel.

As a consequence of our approach, the only idle time that shows up within an episode is due to device or communication latencies (hard idle-time) and cannot be removed by performance changes of the processor. Soft idle-time, on the other hand, occurs between episodes and is mostly due to latencies inherent in user interactions. This type of idle time can be reduced by slowing down the execution of episodes.

3.1. Examples

Perhaps the easiest way to understand what episode detection accomplishes is to take a look at figure 2. This figure shows three execution traces, where the different types of episodes are highlighted in different colors and an abbreviation of the episode type is shown next to a few key episodes (IE – interactive episode, PE – producer episode, and CE – consumer episode). The episode classification is exactly the same as it would be during run-time, no postprocessing takes place (based on knowledge of the future) to derive the exact begin and end timestamps of the episodes. In the traces, a vertical bar of unit width represents one millisecond of execution. The vertical length of the line corresponds to the utilization of the processor in that quantum. Each line is colored according to the type of episode during the execution quantum (black, if it is not part of any specific episode). In some cases, especially in trace B, a single vertical line may be made up of multiple
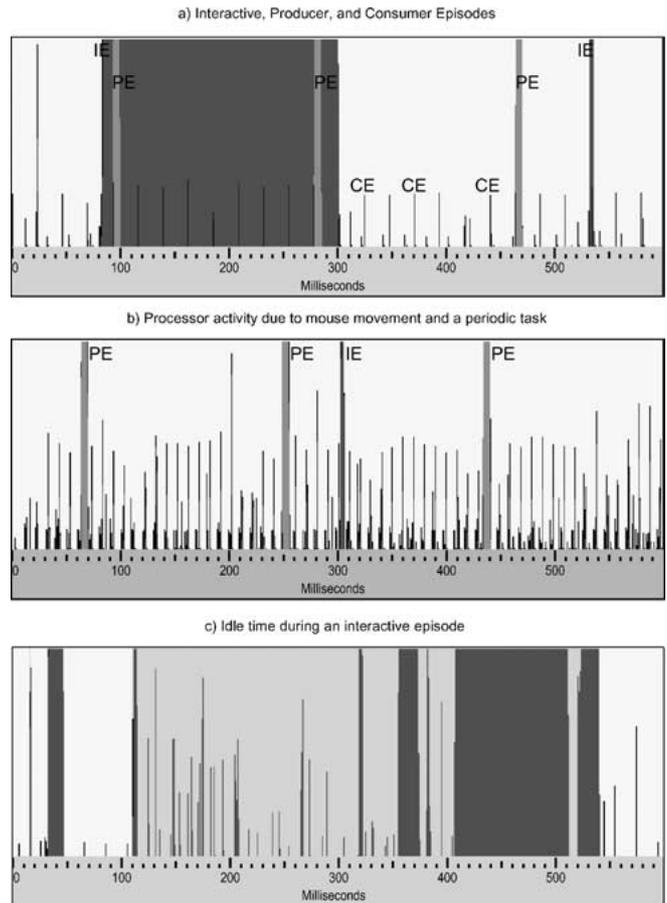
Figure 2. Episodes during execution. The figure shows three execution traces, where the different types of episodes have been highlighted. The first two traces are from the Acrobat Reader benchmark and include an MP3 player running in the background, the third trace is from Netscape accessing web servers on the internet (no MP3 playback in background). IE stands for interactive episode, PE for producer episode, and CE for consumer episode.

episodes, in which case the colors are proportional to the execution lengths of each type of episode during the quantum.

Trace A is representative of execution during interactive applications. It includes two significant interactive episodes along with producer and consumer episodes that are triggered by the MP3 player that is executing on the machine. The first thing to note is that the detection mechanism is not confused by overlapping episodes. The MP3 player kicks in twice (producer episodes) during the first interactive episode, and the classification mechanism accurately attributes the consumed processor time to it. The sound daemon (consumer episodes) wakes up once every 20–23 ms and runs for about a third of a millisecond. During this time it checks whether there is enough data in the sound card and triggers the producer to decode more data when necessary. This behaviour explains why the producer episode is always preceded immediately by a consumer episode. The checking is accomplished by the sound daemon periodically polling the sound device for data requests using the `select` system call. Note that the only code that gets executed during these short periods is the code corresponding to the `select` system call in the kernel, which checks for activity on the monitored devices. The periodicity

of these consumer episodes is determined by how often the kernel schedules processes that are blocked on the `poll` or `select` system calls while waiting for activity. The periodicity can be configured by the caller of the system call but programmers often just rely on the default value in the kernel. When the sound device runs out of data, the `select` call returns – indicating the need to generate more sound data – which in turn wakes up the MP3 player that is blocked in the `write` system call between requests.

This data also illustrates that even a lightweight periodic process can have a significant impact on the user-perceived response time. During the run of the interactive episode, which lasts for 218 ms, the MP3 player and the sound daemon use up about 16 ms of execution time, causing about an 8% increase of the response time.

Trace B shows the bursty activity resulting from the user moving a mouse over the screen, where a document in Acrobat Reader is displayed. In this trace, one sees many short interactive episodes instead of the long ones in trace A. Just as before, the periodic producer episodes are running, however it is more difficult to visually distinguish the consumer episodes from the interactive episodes. There is only one long interactive episode (at around 300 ms), but there are interactive episodes in almost every quantum that has a utilization of about a quarter or more. Short interactive episodes are spaced at about 10 ms apart. The distance between these episodes is determined by the X server, which controls the quality of the user experience. Our X server, when it observes rapid mouse movement, increases screen updates to improve the user experience. The interactive episodes are short because the computation required to update the position of the mouse and to redraw affected regions of the screen are very simple. The only heavier-weight interactive episode in this trace lasts for about 3.5 ms is a result of a change in the appearance of the cursor when it passes over a special region of the Acrobat Reader application.

Trace C illustrates a long interactive episode with a high percentage of idle time from a run of the Netscape benchmark. The interactive episode starts at around 35 ms and ends at 468 ms, and it corresponds to a user loading a web page from a server on the internet. The idle time is due to I/O latency while the page is loaded from a remote server. Initially there are only small bursts of activity, mostly dealing with progress updates but as the requested data starts coming in (at around 280 ms), the rendering engine kicks in and starts generating output to the screen. Our episode detection mechanism accurately attributes the entire episode as an interactive episode instead of breaking it into smaller disjoint parts. This example also illustrates a shortcoming of our scheme, which we believe is a fundamental problem with kernel level episode detection: for a user, it might be sufficient to wait until the first screen of data is rendered instead of waiting for the entire web page to be ready. However, without modifying the web browser, there is no way of knowing when the window update is done. One could add further hooks into the web browser to accurately signal if the user is really interested in data at the bottom of the page.

3.2. Implementation

Our episode detection mechanisms was purposely designed to be as autonomous from other parts of the kernel as possible. Incorporating these techniques into an existing kernel requires only a small number of hooks, but most importantly it does not require changes to existing scheduling algorithms and policies. This contrasts with the approach taken by other researchers that treat the performance-setting problem as a twist on existing scheduling algorithms [7,11,15]. These schemes usually require perfect knowledge about episode deadlines, which need to be specified either by the programmer or the user of the system.

Most applications under UNIX communicate using sockets, signals, and pipes. In particular, the X server uses sockets to communicate with its clients. We do not track interactions via other methods such as System V IPC and shared memory since our benchmarks do not use them. By tracking the communications between the tasks, we are able to determine which tasks have an effect on interactive performance. Unlike other operating systems (e.g., Windows NT), Linux does not differentiate between threads and processes. Threads are implemented using regular processes and the `clone` system call. We use the name "task" as a synonym for both threads and processes. The implementation that performs the tracking is as non-invasive as possible. The difficulty was not in the actual implementation but in finding all the parts of the kernel that needed to be tracked. Currently we track communications through the following system calls:

```
kill, pread, pwrite, read, readv, recv,
recvfrom, rcvmsg, send, sendmsg, sendto,
write, writev.
```

We instrumented each of these system calls to emit a trace of the signals, inodes, and sockets that they are accessing. The socket information is output instead of the inode number, when a socket is accessed through an inode. To be able to match read and write requests through socket pairs, we use the socket's pair (`sock → sk → pair`) on a write and the read socket itself on a read event. Currently we track only communications through UNIX sockets since this is the only socket type that is local to the machine. One could extend this methodology to track communications through other types of sockets if the communicating programs are all local to the machine. However, we have seen no need for this extension so far.

The primary reason for tracking signals is that the thread library (LinuxThreads) uses signals to implement synchronization between threads. By looking at the signal activity we can determine how threads communicate through condition variables, mutexes, and locks. The two functions that needed to be instrumented are `handle_signal` and `send_sig_info`. An alternative to this approach would have been to instrument the thread library; however, our current approach is more generic and has lower overhead.

To determine when tasks are blocked on I/O, we instrumented the `schedule` function to record the reason why it

was called. If it is called from a part of the kernel that is related to I/O (such as the `read` and `write` system calls), then we assume that the task is blocked while waiting for an I/O event to complete. Since there is no predefined way in Linux to find which system call caused a transition to the kernel, we instrumented key system calls to put their id in a field of the executing task's `task_struct`. Once execution gets to the `schedule` function, our code looks at this field and outputs the task's reason for giving up time. Our approach uses a mix of static and dynamic kernel patches. As mentioned above, we have augmented a few kernel data structures and added "hooks" to some kernel functions. However, most of the patching is done dynamically by replacing vectors in the system call table with stubs that monitor the application behaviour.

3.3. Interactive episodes

The beginning of an interactive episode is initiated by the user and is usually signified by a GUI event, such as pressing a mouse button or a key on the keyboard. Finding the end of an episode is more difficult since there is no event that automatically gets generated when the computer is done responding. To find interactive episodes, we keep track of the set of tasks that communicate with each other as a result of a user-initiated GUI event. The start of an interactive episode is initiated by the GUI controller (X server in our case) sending a message through a socket to another task. When this happens both the GUI controller and the receiver of the task are added to what we refer to as the task set of the episode. If the members of the task set communicate with non-member tasks, then the as yet non-member tasks are also added. The end of the episode is reached when all the following conditions are met for tasks in the task set:

- None of the tasks are executing.
- Data written by the tasks have been consumed.
- None of the tasks remains unfinished, as a results of being preempted the last time it ran (i.e., all tasks gave up time on their own by blocking in a system call).
- None of the tasks are blocked on device I/O.

Detecting interactive episodes is only the first step towards performance prediction. Section 4.1 describes how the episode's deadline can be found.

3.4. Periodic episodes

Detecting periodic activity is similar to detecting interactive episodes. However, instead of using communications with the X server as the trigger for starting the episode, we base this decision on whether the initiating task is periodic. To detect periodic activity, we keep track of two pieces information for each task:

- Last execution time.
- Length of the n last periods.
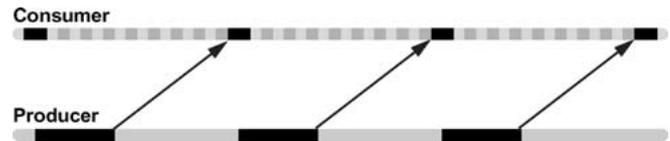
Figure 3. Producer and consumer episodes. The figure shows communications between a producer and a consumer process. The processor can be slowed down to stretch the producer episode to the beginning of the consumer episode.

If a task exhibits only a small amount of variation in period length over the last n runs ($<5\%$), then we treat it as a periodic task.

3.4.1. Producer–consumer episodes

Producer–consumer episodes form a special subcategory of periodic episodes, where the optimum performance level is established by the distance from producer to consumer, not by the distance between periods. A case in point is the Linux esd sound daemon, which wakes up periodically to check for sound playback requests and to send data to the sound card. If esd's playback buffer is not empty, it sends some of the data to the sound card. If the buffer is close to being empty, it wakes up and unblocks the music decoders (e.g., MP3 players), which causes them to generate the next few frames of data.

Figure 3 illustrates interaction between a producer and a consumer process. The distance between the run of the producer and where the data is needed establishes the performance level of the producer. The producer episodes can be stretched to the beginning of their associated consumer episodes. To determine how much the consumer can be slowed down, we first need to determine what the consumer is doing. This information can either be specified by the user on a per-process basis, or one can compute it by observing which devices or processes the consumer communicates with. If the task is communicating with a device, the buffer sizes and the speed requirements of the device establish the minimum speed of the consumer episode. In our current scheme we assume that the consumer can be slowed down to the same speed as the producer.

4. Performance prediction

Our prediction mechanism operates on a per-task basis and uses different algorithms for interactive and periodic episodes. In both cases, the predictor computes the performance factor, which is the ratio of the desired execution speed and the processor's maximum speed.

4.1. Interactive episodes

It is difficult to come up with a good prediction strategy for the optimum performance level of interactive episodes, since interactive episodes are completely dependent on the user, not on some activity within the computer. There is no predictable pattern of recurrence and the lengths of interactive episodes
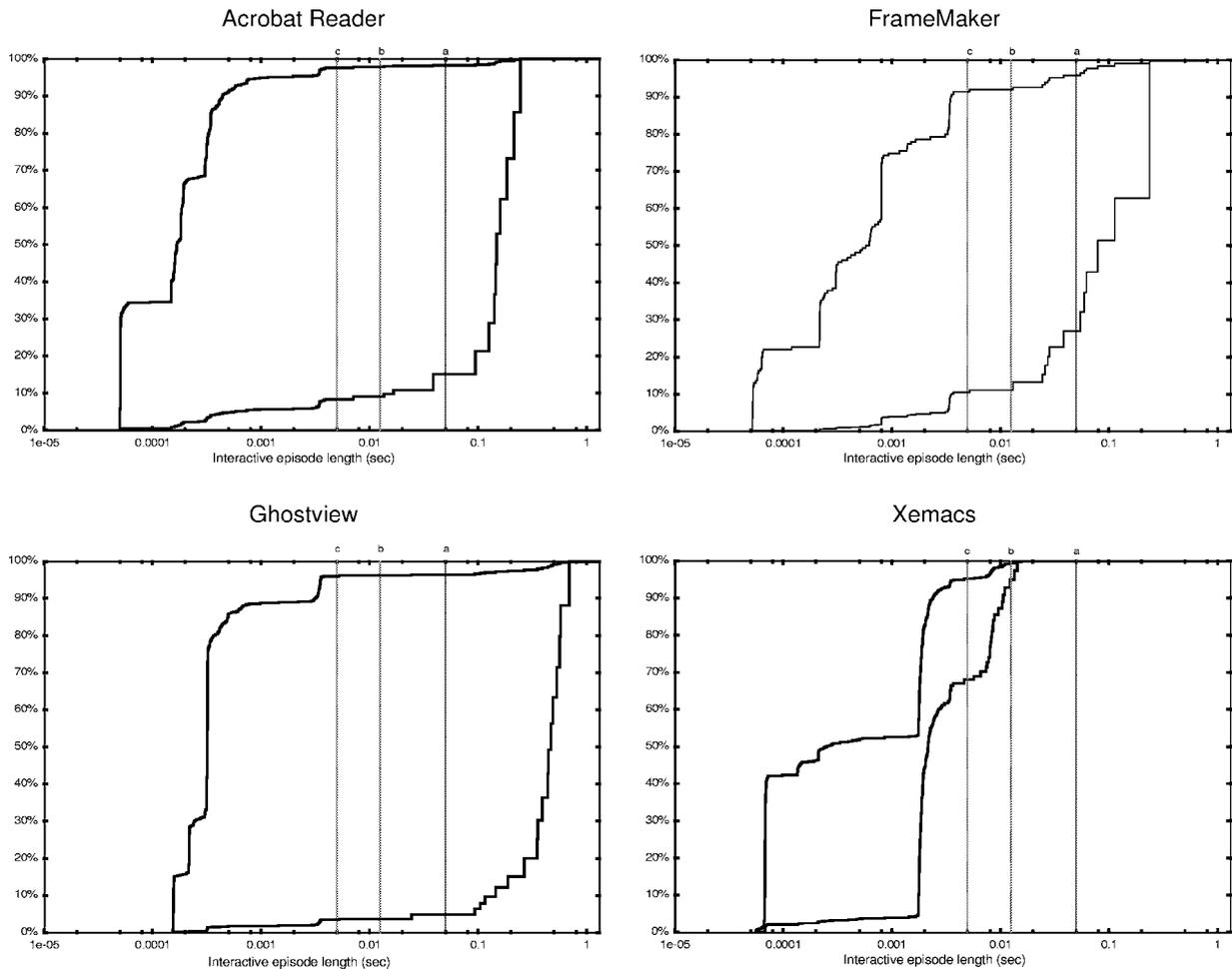
Figure 4. Cumulative interactive episode length distributions. Left line shows the cumulative number, right line the cumulative percentage of time spent in interactive episodes whose lengths are less than or equal to the time specified on the *x* axis. The *x* axis is drawn using a logarithmic scale. Vertical lines *from right to left*: (a) 50 ms, (b) 12.5 ms and (c) 5 ms.

can have orders of magnitude of difference. Our detection scheme allows us to differentiate between different types of episodes (i.e., interactive, producer, consumer) but cannot distinguish between different instances of the same episode in the same task (e.g., when the same button is pushed in the GUI as before).

We believe that the ability to distinguish between interactive episode instances would improve prediction accuracy. However, this would require the kernel to have knowledge about the location in the user program that initiated the given interactive episode. While not impossible, distinguishing the real call-sites from the kernel is difficult to do. A simple comparison based on the user-mode program counter (PC) is not sufficient, since programs usually go through at least one level of indirection (through libC) when calling a system call, and thus all instances of a program's calls to a given system call would have the same user-mode PC. Moreover, since interactive episodes are usually generated as a result of GUI interaction, the necessary number of indirection levels is probably higher due to the use of GUI libraries (e.g., gtk, Xlib). To find the PC value that really distinguishes one interactive episode from another, one would have to chase pointers through mul-

tiple levels, where the actual number of levels depends on the environment (stack layout, libraries, etc.).

Instead of basing a predictor on the ability to distinguish between interactive episode instances, we looked for a simpler solution that only relies on episode type (i.e., interactive, periodic producer or periodic consumer) for prediction.

Figure 4 shows the cumulative distribution of interactive episode lengths for four interactive benchmarks. In each graph, there are two cumulative distributions: the one on the left shows the cumulative number and the one on the right shows the cumulative time spent in interactive episodes of a given length or shorter. To account for the large variation of interactive episode lengths, the time axis is logarithmic. Three vertical lines (a, b, and c from *right to left*) delineate the perception threshold (50 ms), the point under which all episodes finish under the perception threshold at 1/4th of peak performance (12.5 ms), and 1/10th peak performance (5 ms). These values were selected because current processors that are capable of performance and voltage scaling have a minimum performance of about 1/4th peak performance, and future processors could possibly extend the rage of performances to 1/10th of peak value.

These graphs show that while most episodes are very short, the vast majority of time is actually spent in a small fraction that correspond to the long episodes. For example, in Ghostview, 92% of the time is spent in 4% of the episodes. This distribution holds true on the Xemacs benchmark as well, however in this case even the relatively long episodes fall under the perception threshold. Xemacs is an example of an application where one could run almost all of its interactive episodes in the lowest performance level without ever exceeding the perception threshold.

The cumulative episode length distribution graphs imply that a predictor that predicts that an interactive episode only needs the minimum available processor performance would be right more than 90% of the time. However, since these episodes tend to make up only a small percentage of total time – and consequently, have a small contribution to energy use – it is more important to focus on accurately predicting the performance level of the relatively long episodes.

Our performance-factor predictor for interactive episodes works by starting off with an initial performance factor, set to the minimum performance factor of the processor, and then by successively refining its value. Since the initial setting is only relevant for the first interactive episode, the choice of initial value does not have a significant impact on response time or energy savings. The algorithm uses the following three steps:

- Starts running the episode at the predicted performance factor.
- At the end of the episode, computes the duration that corresponds to executing at full performance. Use this information to compute the optimal performance factor for the episode.
- Uses the weighted average of optimum performance factors (PF) as a prediction for future performance factors.

The main observation that we use in our predictor is that it is straightforward to compute what the optimum performance level should have been once an interactive episode is over. During the execution of the episode, the performance-setting of the processor might be changed by external events (e.g., periodic episodes start executing), so the algorithm must keep track of the observed performance factors (pf_i) during the episode's execution. At the end of the episode, this information can be used to estimate how long the episode would have been at full performance:

$$T_{\text{FullSpeed}} = \sum_{i=1}^{n} pf_i(t_i - idle_i) + idle_i. \qquad (1)$$

This equation computes the full speed execution time for an interactive episode given n different observed performance levels during the episode. The variable t_i specifies the length of execution at the ith performance level during an interactive episode, and $idle_i$ is the corresponding amount of idle time.

Based on the estimate of the episode execution time at full performance, the optimum performance level can be estimated for an interactive episode. Equation (2) gives the equation for computing the optimum performance factor for episodes where $T_{\text{FullSpeed}}$ falls between the *minimum-performance threshold* and the *PerceptionThreshold*, where T_{idle} specifies the amount of idle time during the episode:

$$pf_{\text{optimum}} = \frac{T_{\text{FullSpeed}} - T_{\text{idle}}}{PerceptionTreshold - T_{\text{idle}}}. \qquad (2)$$

The minimum-performance threshold specifies the episode duration that could be slowed down to the processor's minimum performance level and still finish under the perception threshold. If the perception threshold is assumed to be 50 ms and the processor's minimum performance is 1/4th of peak, then this value is 12.5 ms. Episodes that are shorter than the minimum-performance threshold can be run at the processor's minimum performance level. Episodes that are longer than the perception threshold need to run at full performance.

We predict the performance factor for the next interactive episode of a given task simply as the average of the optimum performance factors of past interactive episodes, weighted by the duration of each episode:

$$PF_{\text{prediction}} = \frac{\sum_{j=1}^{k} pf_j T_j}{\sum_{j=1}^{k} T_j}. \qquad (3)$$

Equation (3) shows the computation for the predicted performance factor based on the optimum performance factors (pf_j) for k past interactive episodes. T_j refers to the estimated full-speed time of an interactive episode. The size of k can be varied to eliminate saturation and to allow temporal variations of episodes lengths to affect the predictor (see section 4.3).

Since there can be orders of magnitude of difference between the lengths of interactive episodes (see figure 4), this strategy means that the predicted performance factor for short episodes will almost certainly be higher than necessary. This effect is mitigated by the observation that short episodes have only a minimal impact on power consumption.

To recover from prediction errors we set an episode-duration threshold, after which if the episode is still executing, the performance level is raised to full speed. We refer to this threshold as the *PanicThreshold*. While the *PanicThreshold* can ensure that interactive performance does not degrade below a certain level, the goal of the predictor is to set the right performance level at the beginning of the episode, without the need to transition to a higher performance setting later on.

The setting for the *PanicThreshold* reflects the user's tolerance for worst-case performance degradation and determines how speculative the performance factor predictor can be. If the user has no tolerance for possible performance degradation, there is no opportunity for speculation and consequently energy savings. In this case one would be forced to be conservative and always run at full performance to avoid misprediction errors that might extend the episode beyond the perception threshold.

Equation (4) shows the computation for the *PanicThreshold* for a given performance factor (PF) and perception threshold:

$$PanicTreshold = PerceptionTreshold(1 + PF). \qquad (4)$$

This formula allows a longer panic threshold when the initial performance setting is high, because in those cases more work actually gets done per unit time and therefore the cost of an incorrect setting (in terms of its impact on the user) is lower. We also make the assumption that the user allows more performance degradation for episodes whose lengths are close to the perception threshold than for longer episodes. In the worst case, if an episode were to be exactly 50 ms at full speed, then its length will be stretched to 97 ms given a performance factor of 1/4. On the other hand given the same performance factor, an episode that would have been 200 ms at full speed would only be stretched to 247 ms.

4.1.1. The perception threshold

In this paper we use a range of perception thresholds during some of our experiments. Our motivations for this are twofold:

- The higher perception thresholds allow us to estimate the energy and interactive characteristics on a future, higher-performance processor. The 100 ms threshold on today's processor roughly corresponds to the 50 ms threshold on a processor with twice the performance.

- The perception threshold varies by individual and task and may be used as a user settable indicator for his preference for high-performance or energy savings.

Literature about human–computer interaction [1,12] indicates that 20–30 frames per second are sufficient for the human visual system to perceive the images as a continuous stream. This suggests that the perception threshold is around 50 ms. Human subject tests in [1] show that perceptual causality – when two events are perceived to be fused together – ends around 100 ms, and for some test subjects quality degradation begins at around 50 ms.

Other experiments have shown that for simple operations, such as dragging an object through the screen, as few as 5 updates per second are sufficient to maintain an interactive feel (200 ms perception threshold). For non-continuous operations, as much as 1–2 s delays are acceptable [12]. However,

when human motor operations form a feedback loop with visual activity, then it is more important to have a faster response time.

4.2. Incorporating periodic episodes

The optimum performance factor for periodic activity can be computed easily by either stretching the periodic episode's execution to the beginning of the next episode or to the beginning of the associated consumer episode. Since periodic (such as video or sound playback) applications sometimes adjust the quality of playback based on processor performance, it is important to switch to full performance when a periodic application starts executing, so that it has a chance to adapt to the highest performance level. Our assumption is that the user's emphasis is on service quality over energy savings. Others have addressed the trade-off where service quality can be reduced to save energy [4].

An important consideration is to find the performance factor when interactive episodes are present in addition to the periodic activity. Our strategy is very simple:

- When there is no interactive episode executing on the processor, we set the performance factor to the one computed for the periodic activity.

- At the beginning of an interactive episode we switch to the performance factor that was predicted for the task's interactive episodes, if it is higher than the periodic performance factor.

Figure 5 illustrates this strategy during two runs of the Acrobat Reader benchmark. When there is no periodic activity, performance is determined only by the prediction for interactive episodes. However, when periodic activity is present, the algorithm switches between the two performance levels, causing significantly more performance transitions. The spikes that transition the processor to full performance are triggered by interactive episodes whose lengths exceed the *PanicThreshold*. Aside from the initial start at full performance when MP3 is executing in the background, there is
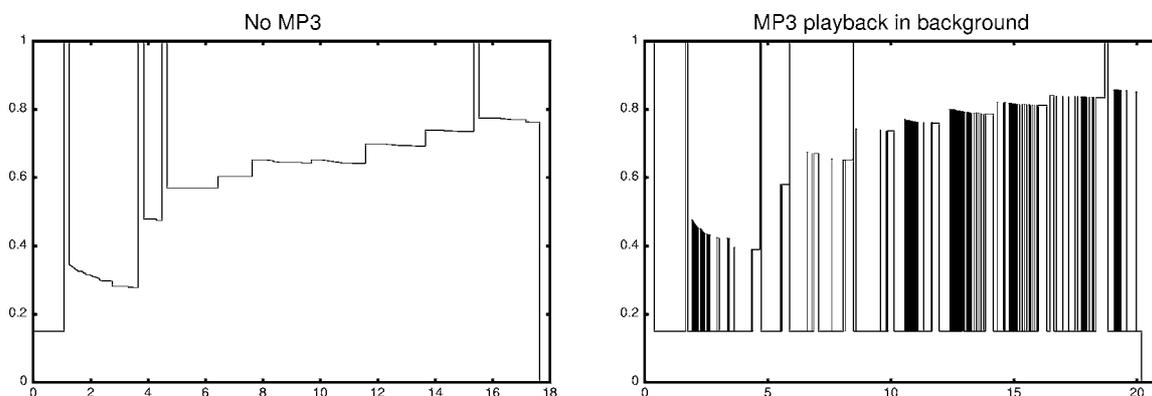

Figure 5. Performance factor settings during the execution of the Acrobat Reader benchmark. Two runs of the Acrobat Reader benchmarks are shown side by side with and without MP3 playback during the run. Perception threshold was set to 200 ms, and data was generated using our simplest strategy (Basic predictor and XSB model without quantization, see section 6.2). More sophisticated models have significantly fewer performance-level transitions when MP3 is executing in the background. Spikes to full performance represent instances when the *PanicThreshold* was reached. Note that the graphs do not show transitions to sleep mode.

only one extra transition due to reaching the *PanicThreshold* on the second figure.

Periodic episodes have the effect of extending the runtimes of interactive episodes [3], which means that the interactive performance-factor predictor should be updated when periodic activity starts. Instead of making the prediction formula more complicated, our approach allows the performance predictor to quickly adapt to the presence of the periodic activity. The next section describes how this is done exactly.

4.3. Implementation details of the Basic predictor

The main features of the Basic predictor are summarized below:

- Interactive episode performance level prediction based on optimum performance factors of past episodes. Estimates of the optimal performance factors are computed at the end of each interactive episode.

- *PanicThreshold* bounds worst-case performance.

- Periodic performance level computed by observing periodic episodes and their communication patterns.

- Switches between periodic and interactive performance factors depending on which episode is executing.

An attractive feature of the predictor is that it requires very little state. We use two variables per task: one keeps track of the sum of episode length weighted performance factors (*totalPF*) and the other keeps track of the total time spent in episodes (*totalTime*). In both cases time is the estimated full-speed execution time of the episode. The performance factor prediction is thus *totalPF* divided by *totalTime*.

One problem with an averaging based predictor is that if the execution time is long, then temporal variation may not influence the predictor for a very long time. One way of alleviating this problem is periodically rescaling the variables by dividing them both by the same amount. This way the predictor can better accommodate a changing workload.

Performance prediction for interactive episodes in the presence of periodic activity relies on rescaling to allow the predictor to adapt to the changing workload. Our studies in [3] have shown that even lightweight background activity, such as MP3 playback, extends the duration of perceptible interactive episodes by an average of 14%. This implies that performance factors predicted based on data without the background activity would underestimate the necessary performance. To alleviate this problem, when periodic activity is detected, the *totalTime* variable is set to 100 ms and *totalPF* is recomputed based on the new value. While providing a reasonable initial prediction, this change allows new performance factor data to take hold quickly.

5. Simulation methodology

Our simulator is driven by traces collected using a modified Linux kernel (2.3.99-pre3) running on a Dell Precision workstation 410, with only one of the two Pentium II 450 MHz processors enabled (512M RAM). The software environment was Mandrake Linux 7 with Helix Gnome 1.2. The traces used in this study are the same as the uniprocessor traces used in [3]. All benchmarks were run by a live user. While we have collected multiple runs in each configuration, in this paper we only use a single run for each simulation. We aimed to repeat each run with MP3 in the background as accurately as possible, but there are slight variations between the runs. All the significant events (e.g., mouse clicks, text entry) were performed in the same order during each benchmark run. However, the exact path of mouse movement (and therefore the interactive episodes corresponding to them) and the amount of time between events varies from one run to the other.

The traces include all significant OS events during the benchmarks execution: thread swap events, system calls, and task information (e.g., name, pid, etc.). Based on this information, our simulator can reconstruct the communication events between the tasks (which imply the synchronization points between them) and simulate the effects of performance scaling. The upside of our methodology is that we have the flexibility to investigate a wide set of architectural parameters. The downside is that the absence of actual hardware prevents us from measuring total energy consumption and from calibrating our results.

6. Energy and performance implications

Our aim is to develop a performance scaling technique that can guarantee that user-perceived performance does not degrade below a user-settable level. A detailed microarchitecture-level power analysis is beyond the scope of this paper; however, we can derive some estimates regarding the expected energy savings using a few simple assumptions.

The metric we use is the energy factor, which is the ratio of the energy used by the scaled workload divided by its predicted energy use at full performance. Equation (5) gives the energy factor formula for a given workload, assuming that the workload is divided into n pieces that execute at the scaled voltage (v_i) and frequency (f_i, specified in MHz) for the scaled amount of time (t_i, in s). T refers to the total execution time at full speed, while the max subscript refers to the maximum value of the given variable:

$$EnergyFactor = \frac{\sum_{i=1}^{n} v_i^2 f_i t_i}{v_{max}^2 f_{max} T}. \quad (5)$$

Our model focuses on the CPU alone and does not take the power consumption of other devices (such as memory and peripherals) into account.

6.1. Processor and power scaling model

Our performance model is based on assumptions from the Intel XScale microarchitecture [2]. Our traces were collected on a 450 MHz Pentium II based workstation, and we make the simplifying assumption that these traces correspond to the full-speed performance on each of the simulated models.

Table 1

Frequency–voltage pairs in our energy models. The table shows the given frequency–voltage pairs of our models. Dashes represent frequency levels that are not supported in a given model.

Model	Frequency (MHz)						
	150	333	400	600	773	800	1000
XSBase	–	1	1.1	1.3	1.5	–	–
XSA	0.75	1	1.1	1.3	1.5	–	–
XSB	0.75			1.2		1.4	1.75

We assume that for each performance transition, there is a 20 ms pause, during which the processor does not execute any instructions. This pause is due to the time it take to re-synchronize the PLLs for the changed frequency value. After this, the performance transition time – during which the voltage level is changed – is assumed to be 1 ms regardless of starting and ending performance level. During this time, we assume that the processor is executing instructions at the rate corresponding to the lower of the two performance levels, but energy is being consumed at the higher.

Table 1 shows the known frequency–voltage values that we used to compute voltage equations (equations (5)–(7)) for arbitrary frequency levels between the minimum and maximum frequencies. The XSBase model corresponds closely to the high-end XScale part (80200M733) described in [2]:

$$v_{\text{XSBase}} = -5 \times 10^{-8} f^2 + 0.0012 f + 0.6261. \quad (6)$$

However, since this model only has a $2.32\times$ frequency range, we extended it to $5.15\times$ by allowing it to go as low as 150 MHz in the XSA model:

$$v_{\text{XSA}} = -4 \times 10^{-7} f^2 + 0.0015 f + 0.5324. \quad (7)$$

The XSB models the parameters of a high-end device that is a research prototype. This processor can vary its performance between 150 MHz and 1000 MHz for a $6.67\times$ swing:

$$v_{\text{XSB}} = 5 \times 10^{-7} f^2 + 0.0005 f + 0.6624. \quad (8)$$

We use the energy factor as our metric for computing energy reduction. It is the ratio of a given workload's energy consumption using our performance-setting strategy over its energy consumption using the processor's peak performance. In all our energy calculations, we assume that the OS power manager puts the processor into a low power sleep mode immediately when no instructions are executing. We do not attribute a power cost to this operation and assume that it happens instantaneously.

During our evaluation we specify a quantization factor for each of the power models. On an actual processor not all frequency–voltage pairs can be directly set up, one must choose from a set of predetermined values. This means that when the performance estimator requests a given performance setting, the actual performance value is rounded up to the next quantum. In our experiments we mostly focus on models where frequency is quantized at 5% steps (100%, 95%, 90%, etc.). We denote quantized models with the suffix 'q' followed by the quantum size.

6.2. Performance and energy characteristics

In this section we examine the characteristics of the basic performance setting algorithm and propose some improvements. Our evaluation focuses on the following three main goals:

- Minimizing the number of performance-level transitions.
- Minimizing the amount of increase in the duration of perceptible interactive episodes.
- Maximizing energy reduction.

These three aspects are closely interrelated. Reducing the number of performance-level transitions is important, because each transition has a delay and energy cost that negatively affects both the perceptible performance and energy savings. On the other hand, increasing the interactive episode duration has a positive effect on energy savings, because the longer interactive episodes may stretch, the slower they can run. However, this has a negative impact on the user-perceived performance. While the increase in perceptible interactive episode duration in all cases falls within the acceptable range (since ensuring this is part of the performance-setting algorithm's job, see section 4.3), we seek to minimize it, i.e., our methodology favors faster response time over energy savings.

The perceptible interactive episode-length increase is computed for all scaled episodes that fall above the perception threshold by dividing the scaled episode length by either the full-speed episode length or the perception threshold, depending on whether the original episode length was longer or shorter (respectively) than the perception threshold.

Table 2 shows our baseline results using the XSB model (without quantization) and 50 ms perception threshold and assumptions described in section 6.1. The mean perceptible interactive episode length increase in all cases is under 30%. Applications that have many short episodes (e.g., Xemacs and Netscape) tend to have the largest increase, while workloads with long episodes (e.g., Ghostview and GIMP) exhibit the smallest increase. This makes sense given that our acceptable delay function (*PanicThreshold*) allows more performance degradation for shorter episodes than for longer ones. One should also note that the number of performance transitions increases significantly (up to four times) when MP3 playback is running in the background. This is because, when a periodic episode is running, the performance setting algorithm alternates between the setting for interactive episodes and the setting for the periodic task. The energy factor tends to be lower when MP3 is running than without. The reason for this is that in most cases the MP3 player requires a lower performance-setting than the interactive application. Xemacs is an exception: the benchmark's interactive episodes require a lower performance-setting than the MP3 player.

Table 3 illustrates the effects of quantization on the results. This model corresponds more closely to an actual hardware implementation than the one used in the previous table. Quantization tends to slightly reduce energy savings and also reduce perceptible interactive episode lengths. The only benchmark where this was not the case is Xemacs, where quantization corrects a few mispredictions, causing both an increase

Table 2
Performance characteristics (XSB, 50 ms perception threshold).

Benchmarks	No MP3 in background				MP3 playback in background			
	Performance transitions	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor	Performance transitions	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor
Acrobat Reader	543	13%	7%	0.91	668	13%	11%	0.84
FrameMaker	155	20%	11%	0.89	191	9%	7%	0.75
Ghostview	510	5%	1%	0.98	1149	5%	1%	0.91
GIMP	919	5%	4%	0.97	1731	5%	4%	0.91
Netscape	1026	18%	14%	0.87	1739	21%	12%	0.82
Xemacs	381	23%	20%	0.30	1417	29%	33%	0.34

Table 3
Performance characteristics (XSBq5, 50 ms perception threshold).

Benchmarks	No MP3 in background				MP3 playback in background			
	Performance transitions	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor	Performance transitions	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor
Acrobat Reader	28	12%	5%	0.92	664	12%	10%	0.86
FrameMaker	15	17%	11%	0.90	184	7%	6%	0.77
Ghostview	10	4%	0%	0.99	1135	4%	0%	0.92
GIMP	28	4%	3%	0.98	1533	5%	3%	0.92
Netscape	32	17%	12%	0.88	1547	20%	11%	0.84
Xemacs	15	15%	14%	0.26	1416	29%	31%	0.32

Table 4
Performance characteristics with MP3 playback in background (XSBq5, 50 ms perception threshold).

Benchmarks	MP3 IEPerf transition start latency = 1 ms				MP3 IEPerf transition start latency = 5 ms			
	Performance transitions	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor	Performance transitions	Mean perceptible IE length increase	Median perceptible IE length increase	Energy factor
Acrobat Reader	637	11%	4%	0.86	125	13%	6%	0.75
FrameMaker	153	8%	7%	0.76	81	12%	9%	0.73
Ghostview	1031	5%	1%	0.91	222	6%	1%	0.86
GIMP	854	5%	4%	0.88	334	6%	6%	0.83
Netscape	1072	18%	12%	0.83	340	20%	14%	0.72
Xemacs	1047	29%	32%	0.32	980	34%	39%	0.31

in energy savings and a decrease in the average perceptible episode length.

Perhaps the most striking improvement over the data in table 2 is the dramatic reduction in the number of performance-level transitions (in some cases more than 300-fold) when no MP3 playback is running concurrently with the interactive application. This behaviour is also due to the fact that when there is no periodic background activity, the successively predicted performance levels are close to each other, causing quantization to eliminate the minor corrections. When MP3 playback is present, the deliberate transitions between interactive and periodic modes keeps the number of transitions high.

The number of performance transitions can be further reduced based on an observation of figure 4. We have pointed out that the majority of interactive episodes are very short (less than 1 ms) and that very little time is spent in those episodes (<5%). When there is no periodic background activity, the effect of the short interactive episodes is negligible since the performance level is always set at the pre-

dicted interactive performance level, however, when there is an MP3 player in the background, these short interactive episodes cause an unnecessary transition from the periodic to the interactive performance level. When the interactive episode is very short, this transition simply wastes energy, since the episode is likely to be finished before the transition is over.

This observation suggests a strategy that waits for a certain amount of time before starting a transition to the interactive performance level. Table 4 illustrates the effects of a transition-start latency before interactive episodes. The data is only shown for the benchmarks with MP3 playback in the background, because there was no significant change in results when background activity was not present.

Contrasting with table 3 shows that a 1 ms transition-start latency leaves the energy factors mostly unchanged but causes a small reduction in the number of performance transitions. Extending the transition-start latency to 5 ms causes both a significant reduction in energy consumption and in the number of performance transitions (up to a 5-fold reduction).

No MP3   MP3 playback in background
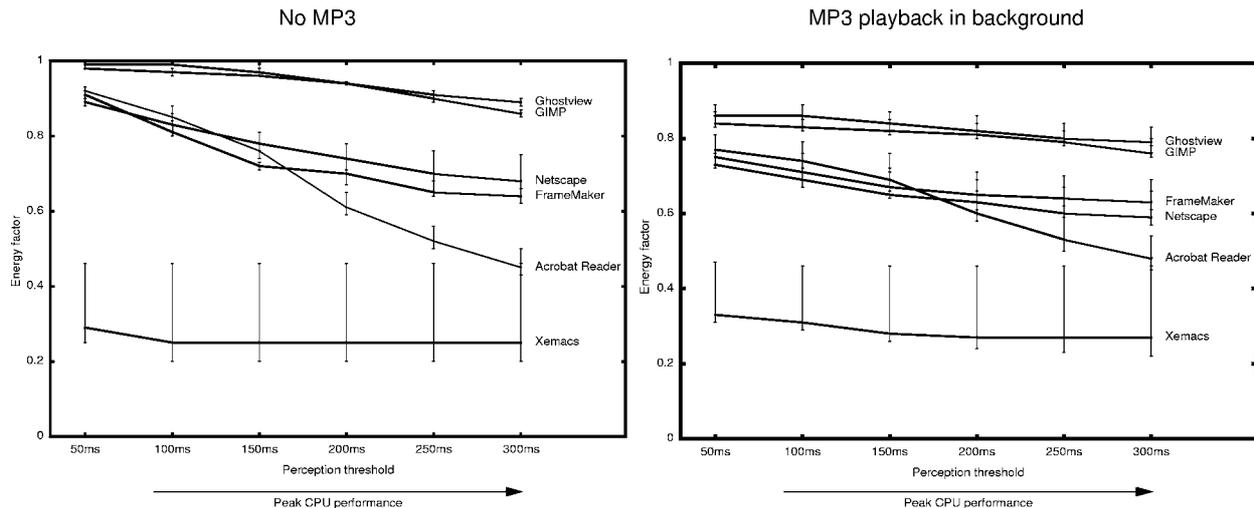
Figure 6. Energy factors corresponding to different perception thresholds using three quantized (5%) models. These graphs show results using the Enhanced predictor corresponding to a variety of perception thresholds. At each perception threshold level, we show the energy factors for the QSBaseQ5 (top point), XSAq5 (middle point, connected) and XSBq5 (bottom point) models.

Moreover, the average perceptible interactive episode-lengths stay at around the same level as before.

6.3. The Enhanced predictor

The previous section suggests two minor changes to the Basic predictor: (1) to quantize the allowable performance levels; and (2) to wait for a certain time before changing the performance factor when an interactive episode starts. For simplicity we only use a statically specified transition start latency of 5 ms in the Enhanced predictor. A more sophisticated predictor could dynamically compute a per-process value.

Figure 6 shows the energy factors using the Enhanced predictor and the XSBaseQ5, XSAq5 and XSBq5 power models given a variety of perception thresholds. Our results show that while on the measurement machine there is little opportunity for power savings, as the peak performance of the processor gets faster, energy savings will be more pronounced. We must note that our traces were collected on a 450 MHz Pentium II machine, and today's high-end processors are already 2–3 times faster. We estimate that the energy factor on today's high-end desktops could be in the 10–75% range at the 50–100 ms perception threshold.

In the figure, energy factors corresponding to the XSAq5 model are connected by a line at each perception threshold, the results for the XSBaseQ5 and XSBq5 are shown as the bars above and below each point (respectively). In all cases, XSBq5 achieves the largest energy savings, while XSBaseQ5 achieves the least. For most applications the difference between the three models is small: in most cases less than 5%. However, the difference is significant on the Xemacs benchmark runs, since this application spends most of its time at the lowest performance setting allowed in each model. In this case, the lower minimum performance levels of the XSA and XSB models give them a significant edge.

Periodic activity has the effect of vertically compressing the graph towards its center. The load due to the background activity increases the energy savings for benchmarks with long interactive episodes. On the other hand, one can observe the exact opposite effect when the interactive episodes are short (Xemacs).

6.4. Desired hardware improvements

We have already shown that by reducing the number of performance-level changes, quantization can have a positive effect on both energy savings and interactive performance. In our measurements we found that the quantum size does not greatly impact our results. Using the Enhanced predictor, the difference in energy savings due to quantum sizes of 5%, 10%, and 20% are negligible. While, in all cases the 5% quantum size has the smallest energy factor, the 20% quantum size is only behind by at most 1%. Using Oracle information, the difference in energy savings due to different quantum sizes would always be under 5%.

For all our measurements we assume that there is a 20 μs pause when a performance transition is initiated, and that a transition takes 1 ms. The current pause duration, which we believe is indicative of what can realistically be expected, is short enough that eliminating it has neither a significant impact on energy savings, nor on perceptible interactive performance. However, there might be other reasons for lowering it, such as latencies incurred during communication with peripheral devices.

Reducing the lengths of performance transitions, on the other hand, has a positive effect on both perceptible performance and energy savings. While shortening performance transitions has an overall positive effect, a better prediction mechanism (as shown by the Oracle predictor) could achieve even more significant improvements. We believe that the Enhanced predictor could be improved by giving it the ability to distinguish between episode instances (as discussed in section 4.1), not just episode types.

7. Conclusions and future work

In this paper we describe an automatic episode detection mechanism that can be used to guide the performance-setting decisions for a processor that supports dynamic performance and voltage scaling. This system can derive and predict episode deadlines automatically, without the need to modify existing user programs. We have shown that our approach can achieve significant energy savings while ensuring that interactive performance stays at an acceptable level. We are currently working on the evaluation of our algorithms on a system that is capable of dynamic voltage scaling.

While our current implementation is tied closely to the Linux kernel and its application environment, we believe that the ideas proposed in this paper are also applicable to other operating systems. We developed our methodology for Linux by observing common program design and communication patterns. While the specifics may vary from one OS to another, most modern operating systems have abstractions that a similar monitoring environment could be built on (i.e., inter-process communication, multithreading, system calls).

Our mechanism works without modifications of user programs, however an optional API might be useful for applications that want to take full advantage of performance scaling. One of the biggest shortcomings of the current predictor is its inability to distinguish episode instances from one another. An API that would allow the programmer to delineate and name critical episodes and perhaps optionally specify its type and deadline would help. The API might consist of the following system calls:

```
episode_begin <id> [type][deadline]
episode_end <id>
```

The `id` is a per-task identifier assigned by the programmer to distinguish one episode from another. The `type` field optionally categories. The `deadline` field optionally specifies the maximum length of the episode. The idea behind this API is that its main role is to give hints to the existing prediction and communication-tracking mechanism, instead of superseding it. For example, there is no need to specify dependencies between episodes since that information can be derived automatically from information in the kernel.

We have shown that along with peak performance, it is also important to allow the processor to run slowly. While there will always be applications that can only run acceptably at the processor's fastest setting, an increasing number of applications are able to take advantage of the lower performance modes of the processor. As peak performance of the processor increases, it is important to widen the gap between the minimum and maximum performance levels of the processor.

We believe that the core idea of our technique – on-line monitoring and dynamic adaptation – could be extended to allow the kernel to make better scheduling and service-quality decisions.

Acknowledgements

References

[1] S.K. Card, T.P. Moran and A. Newell, *The Psychology of Human–Computer Interaction* (Lawrence Erlbaum Associates, 1983).

[2] Developer manual, Intel 80200 processor based on Intel XScale microarchitecture, http://developer.intel.com/design/iio/manuals/273411.htm

[3] K. Flautner, R. Uhlig, S. Reinhardt and T. Mudge, Thread-level parallelism and interactive performance of desktop applications, in: *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)* (November 2000).

[4] J. Flinn and M. Satyanarayanan, Energy-aware adaptation for mobile applications, in: *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP-17)* (December 1999).

[5] K. Govil, E. Chan and H. Wasserman, Comparing algorithms for dynamic speed-setting of a low-power CPU, in: *Proceedings of the First International Conference on Mobile Computing and Networking* (November 1995).

[6] D. Grunwald, P. Levis, K. Farkas, C.B. Morrey III and M. Neufeld, Policies for dynamic clock scheduling, in: *Proceedings of the Fourth Symposium on Operating Systems Design & Implementation* (October 2000).

[7] C.M. Krishna and Y.-H. Lee, Voltage-clock-scaling adaptive scheduling techniques for low power hard real-time systems, in: *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)* (2000).

[8] D. Laird, Crusoe processor products and technology (January 2000) http://www.transmeta.com/press/download/pdf/laird.pdf

[9] J. Lorch and A.J. Smith, Improving dynamic voltage scaling algorithms with PACE, in: *Proceedings of the ACM SIGMETRICS 2001 Conference* (June 2001).

[10] T. Mudge, Power: A first class design constraint for future architectures, in: *Proceedings of the 7th International Conference on High Performance Computing (HiPC 2000)* (December 2000).

[11] T. Okuma, T. Ishihara and H. Yasuura, Real-time task scheduling for a variable voltage processor, in: *Proceedings of the International Symposium on System Synthesis* (November 1999).

[12] D.R. Olsen, *Developing User Interfaces* (Morgan Kaufmann, 1998).

[13] T. Pering, T. Burd and R. Brodersen, The simulation and evaluation of dynamic voltage scaling algorithms, in: *Proceedings of International Symposium on Low Power Electronics and Design 1998* (June 1998) pp. 76–81.

[14] T. Pering, T. Burd and R. Brodersen, Voltage scheduling in the lpARM microprocessor system, in: *Proceedings of the International Symposium on Low Power Electronics and Design 2000* (July 2000).

[15] Y. Shin and K. Choit, Power conscious fixed priority scheduling for hard real-time systems, in: *Proceedings of the 36th Annual Design Automation Conference* (1999).

[16] M. Weiser, B. Welch, A. Demers and S. Shenker, Scheduling for reduced CPU energy, in: *Proceedings of the First Symposium of Operating Systems Design and Implementation* (November 1994).

Krisztián Flautner is currently a Principal Research Engineer at ARM Limited. He received a Ph.D. degree in computer science and engineering from the University of Michigan in 2001. His thesis explored the relevance of multithreading for interactive desktop workloads and described the implementation of an automatic power-management algorithm for processors supporting dynamic voltage scaling. In the research group at ARM, he is currently working on the next generation ARM architecture.

E-mail: krisztian.flautner@arm.com

Steven K. Reinhardt received the B.S. degree in electrical engineering from Case Western Reserve University in 1987, the M.S. degree in electrical engineering from Stanford University in 1988, and the Ph.D. degree in computer science from the University of Wisconsin-Madison in 1996. He is currently an Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan in Ann Arbor. His primary research interest is in computer system architecture, focusing on uniprocessor and multiprocessor memory systems, multithreaded systems, and system simulation techniques. He is the recipient of a 2001 Sloan Research Fellowship and a 1998 NSF CAREER award. Dr. Reinhardt is a member of ACM, IEEE, and the IEEE Computer Society.

E-mail: stever@eecs.umich.edu

Trevor Mudge received the B.Sc. degree from the University of Reading, England, in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively. Since 1977, he has been on the faculty of the University of Michigan, Ann Arbor. He is presently a Professor of Electrical Engineering and Computer Science. He recently concluded a ten years term as the Director of the Advanced Computer Architecture Laboratory – a group of eight faculty and about 70 graduate students. He is the author of numerous papers on computer architecture, programming languages, VLSI design, and computer vision. He has also chaired about 30 theses in these research areas. His research interests include computer architecture, computer-aided design, and compilers. In addition to his position as a faculty member, he runs Idiot Savants, a chip design consultancy, and he is a technical advisor to several venture firms. Trevor Mudge is a Fellow of the IEEE, a member of the ACM, the IEE, and the British Computer Society.

E-mail: tnm@eecs.umich.edu