Chameleon: Application Level Power Management with Performance Isolation

Xiaotao Liu
Computer Science Dept.
University of Massachusetts
Amherst, MA 01003
xiaotaol@cs.umass.edu

Prashant Shenoy
Computer Science Dept.
University of Massachusetts
Amherst, MA 01003
shenoy@cs.umass.edu

Mark Corner
Computer Science Dept.
University of Massachusetts
Amherst, MA 01003
mcorner@cs.umass.edu

ABSTRACT

In this paper, we present Chameleon—an application-level power management approach for reducing energy consumption in mobile processors. Our approach exports the entire responsibility of power management decisions to the application level. We propose an operating system interface that can be used by applications to achieve energy savings. We consider three classes of applications—soft real-time, interactive and batch—and design user-level power management strategies for representative applications such as a movie player, a word processor, a web browser, and a batch compiler. We also design a user-level power manager based on *GraceOS* using Chameleon. We implement our approach in the Linux kernel running on a Sony Transmeta laptop. Our experiments show that, compared to the traditional system-wide CPU voltage scaling approaches, Chameleon can achieve up to 32-50% energy savings while delivering comparable or better performance to applications. Further, Chameleon imposes small overheads and is very effective at scheduling concurrent applications with diverse energy needs.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Algorithms, Design, Experimentation

Keywords

Power Management, Mobile Computing, Multimedia

1. INTRODUCTION

Recent technological advances have led to a proliferation of mobile devices such as laptops, personal digital assistants (PDAs), and cellular telephones with rich audio, video, and imaging capabilities. While the processing, storage, and communication capabilities of these devices have improved significantly, these advances have outpaced the improvements in battery capabilities. Consequently, energy continues to be a scarce resource in such devices. The situation is exacerbated by the resource-hungry nature of many applications, such as movie players and batch compilations.

Modern mobile devices use energy judiciously by incorporating a number of power management features. For instance, modern processors such as Intel's XScale and Pentium-M and Transmeta's Crusoe incorporate dynamic voltage and frequency scaling (DVFS). DVFS enables dynamically variable CPU speed which can reduce energy consumption during periods of low utilization [11, 12, 24]. In general, such techniques must be carefully designed to prevent the processor slowdown from degrading the responsiveness of applications.

This paper explores a new approach, namely application-level power management. We argue that applications know best what their resource and energy needs are, and consequently, applications can implement better power management policies than the operating system. We propose an approach where applications are given complete control over their CPU power settings—an application is allowed to specify its CPU power setting independently of other applications, and the operating system isolates an application from the settings used by other applications. Our approach resembles the philosophy of the *Exokernel*, where the OS grants complete control of various resources to the applications and only enforces protection to prevent applications from harming one another [5]. The Exokernel project successfully demonstrated the benefits of application-level networking, application-level memory management, application-level file systems and CPU scheduling [5]. Our work extends this notion to application-level power management.

Research Contributions: The notion of application-level power management opens up a realm of possibilities that are infeasible using existing approaches.

- *Performance:* Our approach enables each application to make local power management decisions based on its processor demand and processor availability. We experimentally show that local decisions by individual applications can globally optimize system-wide energy usage and are better than choosing a single system-wide power setting for all applications.
- *Flexibility:* Such an approach enables each application to implement a power management policy that closely matches its energy and performance requirements. Different applications can choose different policies and yet coexist with one another concurrently. Legacy applications or those applications that do not wish to implement their own strategy can delegate this task to a user-level power manager that chooses appropriate settings based on observed behavior.
- *Generality:* Our approach is general and unlike some existing approaches, does not make specific assumptions about

MM'05 November 6–12, 2005, Singapore.

the nature of applications. Any application can use the power management interface to manage its energy needs, and we demonstrate such strategies for several different applications.

- *Modest implementation costs:* We show that user-level power management policies can be implemented at a modest cost. For applications considered in this work, the cost of implementing our policies varied from 40 to 239 lines of code, a relatively minor modification to applications hundreds of thousands of lines of code.

At first glance, it may appear that an application-level power management approach loses the ability to couple the power management strategy with the CPU scheduling algorithm. At least one recent approach has advocated such an integrated approach for power management and scheduling [26, 27]. Contrary to intuition, we show that it is indeed possible to implement such couplings between the scheduler and the power manager using our application-level framework. We demonstrate the feasibility of doing so by implementing *GraceOS* [26, 27] as a user-level power manager in our system. By carefully exporting resource usage statistics from within the kernel and using a flexible power management interface, we show how the power management policy can be implemented in user-space while retaining the ability to interact with the scheduler.
Chameleon, our application-level power management approach consists of three components: (i) a *common OS interface* that can be used by power-aware applications to measure their CPU usages and adjust their CPU speed settings, (ii) a modified kernel *CPU scheduler* that supports per-process CPU speed settings and ensures performance isolation among processes, and (iii) a *speed adapter* that maps these CPU speed settings to the nearest speed actually supported by the hardware.
We consider three classes of applications—soft real-time, interactive, and batch—and show how soft real-time applications such as movie players, interactive applications such as word processors and web browsers, and batch applications such as "make" can each implement a different power management strategy. We specifically demonstrate *how these applications can coexist concurrently and yet globally optimize system-wide energy consumption.*
We implement a prototype of Chameleon in the Linux kernel and evaluate its effectiveness on a Sony laptop equipped with Transmeta's Crusoe TM5600-667 processor [23]. Our experiments compare Chameleon with three existing OS-level DVFS approaches, namely PAST [24], PEAK [12] and AVG_n [11] and with LongRun, a hardware-based DVFS approach. Our experiments with individual power-aware applications show that Chameleon can extract up to a 32% energy savings when compared to LongRun and up to 50% savings when compared to OS-based DVFS approaches, without any performance degradation to time-sensitive multimedia and interactive applications. Our experiments with *concurrent* applications show that local power management decisions in Chameleon yield 20-50% energy savings over LongRun and OS approaches that use a single power setting for all applications, thereby demonstrating the benefits of allowing each application to use a custom power setting that is most appropriate to its needs.
The rest of this paper is organized as follows. Section 2 presents an overview of the Chameleon. Sections 3 and 4 present our user-level power management strategies. Section 5 discusses our implementation. Section 6 presents our experimental results. Finally, Sections 7 and 8 presents related work and our conclusions.

2. CHAMELEON ARCHITECTURE

Chameleon consists of three key components (see Figure 1). First, Chameleon consists of an *OS interface* that enables applications to

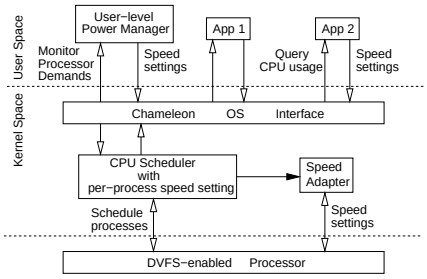


Figure 1: The Chameleon Architecture.

query the kernel for resource usage statistics and to convey their desired power settings to the kernel. The details of the interface are presented in Section 5. In general, a user-level power management strategy combines OS-level resource usage statistics with application domain knowledge to determine a desirable CPU power setting. This can be achieved in two ways. An application can use the Chameleon interface to directly modify its own power settings. Alternatively, an application can delegate the task of power management to a user-level power manager. Such a power manager can use resource usage statistics and any application-supplied information to adjust the application's power settings on its behalf.
Second, Chameleon implements a modified CPU scheduler that supports per-process CPU power settings and application isolation. The scheduler maintains the current power settings for each process and conveys these settings to the underlying processor whenever the process is scheduled for execution (i.e., at context switch time). The application's power settings can be modified at any time via system calls, either by the application itself or by a user-level power manager acting on its behalf. An application's power settings take effect *only when it is scheduled*, and further, applications get the same share of the CPU regardless of their power settings. Consequently, applications are isolated from one another and from the settings used by malicious or misbehaving applications. Kernel support for per-process power settings and application isolation does not require any direct modifications to the CPU scheduling algorithm itself, and as a result, Chameleon is compatible with any scheduling algorithm.
Third, Chameleon implements a speed adapter that maps application-specified power settings to the nearest CPU speed actually supported by the hardware. In particular, an application specifies the desired CPU speed as a fraction f_i of the maximum processor speed. The speed adapter maps this fraction to the nearest supported CPU speed; since different processors support different discrete speeds, such an approach ensures portability across hardware.
Although this work considers applications that manage their own energy needs, in practice, it may not be feasible to modify every single application to make it power-aware. Thus, legacy applications will coexist with power-aware applications in Chameleon. For such applications, Chameleon can either delegate them to a user-level power manager or revert to a hardware DVFS technique. In the former case, the manager determines power settings based on external observations of application behavior. In the latter case, whenever a power-unaware application is scheduled on the CPU, Chameleon dynamically switches to a system-controlled DVFS technique (our current prototype uses LongRun [7]). This hardware DVFS technique is disabled when a power-aware application is scheduled for execution. Both techniques enable legacy applications to extract some power savings while permitting power-aware applications to maximize these savings.

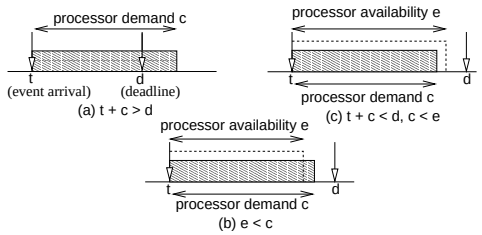


Figure 2: Three scenarios for task execution.

3. APPLICATION-LEVEL POWER MANAGEMENT

Regardless of the actual application, our user-level power management policies consist of three key steps. (i) *Estimate processor demand:* In this step a combination of application domain knowledge and past CPU usage statistics is used to estimate processor demand in the near future. (ii) *Estimate processor availability:* This step explicitly accounts for the impact of other concurrent applications. It estimates the amount of CPU time that will be available to the application in the presence of other applications. (iii) *Determine processor speed setting:* The third step chooses a speed setting that "matches" the processor demand to the processor availability. For instance, if the actual demand is only half of the available CPU time, then the application can run the processor at half speed and spread its CPU demand over the available time. In contrast, if the processor demand and the processor availability are roughly equal, the application may choose to run the processor at full speed.

In the rest of this section, we show how these ideas can be instantiated for four specific applications that belong to three different application classes—soft real-time, interactive best-effort, and batch.

3.1 MPEG Video Decoder

An MPEG video decoder is an example of a soft real-time application. Many multimedia applications such as DVD playback, audio players, music synthesizers, video capture and editors belong to this category. A common characteristic of these applications is that data needs to be processed with timeliness constraints. For instance in a video decoder frames need to be decoded and rendered at the playback rate—in a 25 frames/s video, a frame needs to be decoded once every 40ms. The inability to meet timeliness constraints impacts application correctness; playback glitches will be observed in a video decoder, for example.

A soft real-time application can use the following general strategy for user-level power management. Assume that the application executes a sequence of tasks; the decoding of a single frame is an example of a task. Let c denote the amount of CPU time needed to execute this task at full processor speed. Let d denote the deadline of this task and let t denote the task begin time. Further, let e denote the amount of CPU time that will actually be allocated to the application for this task before its deadline. The parameter c captures processor demand, while e captures processor availability by accounting for the presence of other concurrent tasks in the system. In a time sharing scheduler, for instance, the larger the number of runnable tasks, the smaller the value of e. In a QoS-aware scheduler that allows a fixed fraction of the CPU to be reserved for an application, the value of e will be independent of other tasks in the system.

Given the processor demand c, processor availability e and deadline d, the processor speed can be chosen as follows.

Case 1: If $t + c > d$ then it is impossible to meet the task deadline (see Figure 2(a)). Essentially, the task started "too late," and neither the CPU scheduler not the power management strategy can rectify the situation. In such a scenario, the appropriate policy is to choose the full processor speed for this task.

The next two scenarios assume that case 1 is not true and that it is possible to meet the task deadline.

Case 2: If $e < c$, then the processor demand exceeds processor availability for this task (see Figure 2(b)). Although it is feasible to meet the deadline by allocating sufficient CPU time to the task, the CPU scheduler is unable to do so due to presence of other concurrent applications. Since application performance will suffer due to insufficient processor availability, the power management strategy should not further worsen the situation. Thus, the application should run at full processor speed for this task. Any other strategy would violate our goal of isolation.

The final scenario assumes that neither cases 1 or 2 are true.

Case 3: If $t + c < d$ then task can finish before its deadline at full processor speed (see Figure 2(c)). In this case, the policy should slow down the CPU such that the demand c is spread over the amount of time the task will execute on the CPU, while still meeting the deadline. The CPU frequency f should be chosen as

$$f = \frac{c}{min(e, d - t)} \cdot f_{max} \quad (1)$$

where f_{max} is the maximum processor speed (frequency).

This strategy is applicable to a variety of soft real-time applications, so long as the notion of a task is defined appropriately. In a video decoder, (i) decoding of each frame represents a task, (ii) c denotes the time to decode the next frame at full speed, (ii) e denotes the estimated duration for which the decoder will scheduled on the CPU until the frame deadline, and (iii) d denotes the playback instant of the frame (as determined by the playback rate of the video). While d is known, parameters c and e need to be estimated for each frame.

Estimating processor demand: Processor demand is determined by estimating frame decode times. We consider *MPlayer* an open-source video decoder that supports both MPEG-2 and MPEG-4 playback. Note that MPEG-2 is widely used for DVD playback, while MPEG-4 is used by commercial streaming systems such as QuickTime and Windows Media; *mplayer* is representative of these applications. A number of MPEG-2 and MPEG-4 video clips with different bit rates and spatial resolutions were decoded by an instrumented *mplayer* that measured and logged the decode time of each frame at full processor speed. We analyzed the resulting traces by studying the first and second order statistics of the decode times and frame sizes for each frame type (i.e., I, P, B). Our analysis, the details of which may be found in [13], shows a piece-wise linear relationship between the decode times and the frame sizes for each frame type. These results corroborate the findings of a prior study on MPEG-2 where an approximate linear relationship between frame size and decode times was observed [1]. Using these insights, we constructed a predictor that uses the type and size of each frame to compute its decode time. A key feature of our predictor is that the prediction model is parameterized at run-time to determine the slope and intercept of the piece-wise linear function. To do so, the video decoder stores the observed decode times of the previous n frames, scales these values to the full-speed decode time (since the observed decode times may be at slower CPU speeds), and uses these values to periodically recompute the slopes and the intercepts of the piece-wise linear predictor. This not only enables the predictor to account for differences across video clips (e.g., different bit rates require different linear predictors), it also accounts for variations within a video (e.g., slow moving scenes versus fast moving scenes in a video). The parameterized predictor is then used to estimate the decode time of each frame at full processor speed. Additional details of our predictor including its experimental validation may be found in [13].

Estimating processor availability: Using the Chameleon inter-

face, the application can obtain the start and end times of the previous k instances where the application was scheduled on the CPU. This history of quantum durations and the start times of the quanta provide an estimate of how much CPU time was allocated to the application in the recent past. An exponential moving average of these values can be used to determine the amount of CPU time that is likely to be allocated to the application per unit time, and this yields the processor availability over the next $d - t$ time units.

Determining processor speed: Given an estimate $\hat{c}$ of the frame decode time and $\hat{e}$ of the processor availability, the actual CPU frequency f is chosen in *mplayer* as follows:

$$f = \begin{cases} f_{max} & \text{if } t + \hat{c} > d \\ f_{max} & \text{if } \hat{e} < \hat{c} \\ min(\frac{\hat{c} \cdot f_{max}}{min(\hat{e}, d-t)+\beta}, f_{max}) & \text{otherwise} \end{cases} \quad (2)$$

where β is a correction factor that is used to account for past errors in frame decode times. It the actual decode times are consistently overestimated or underestimated by the predictor, the factor β can be used to correct this error. The Chameleon speed adapter then maps the computed f to the closest supported CPU speed that is no less than the requested speed.

Implementation: We modified *mplayer* to implement the frame decoding time predictor and the speed setting strategy. Our modifications were primarily restricted to the beginning and end of frame decoding method in *mplayer*. We used `gettimeofday` to measure the frame decoding time and the Chameleon interface to estimate the processor availability. Other modifications involved using the Chameleon interface to set the CPU speed using Equation 2. In all, the implementation of frame decoding time predictor involved 221 lines of C code, and the implementation of speed setting strategy involved 18 lines of C code. This indicates that user-level power management strategy can be implemented at modest effort.

3.2 Word Processor

A word processor from an Office suite is an example of an interactive best-effort application. Many applications such as editors, shell terminals, web browsers and games fall into this category. We consider *AbiWord*, a popular open-source word processor from the Gnome Office suite. AbiWord is an event-driven application that works as follows. After an event such as a mouse click or key stroke, the word processor must handle the event. For example, when the user clicks on a menu item, the application must display a drop-down menu of choices. When the user types a sentence, each character representing a keystroke needs to be displayed on the screen. The window needs to be redrawn when the *draw* event arrives. The speed at which these events are processed by the word processor greatly impacts the user's experience.

Studies have shown that there exists a human perception threshold under which events appear to happen instantaneously [2]. Thus, completing these events any faster would not have any perceptible impact on the user. While the exact value of the perception threshold is dependent on the user and the type of task being accomplished, a value of 50ms is commonly used [2, 6, 14, 15]. We also use this perception threshold in our work.

An event-driven interactive application should choose CPU speed settings such that each event is processed *no later than the human perception threshold*. One possible strategy to do so is to (i) estimate the processor demand of an event, (ii) estimate the processor availability in the next 50 ms, and (iii) choose a speed such that the demand is spread over the available CPU time while still meeting the 50 ms perception threshold. Since an event-based application may process many different types of events, estimating processor demand for each event will require the approach to be explicitly aware of different event types and their computational needs. Such a strategy can be quite complex for applications such as browsers or a word processors that support a large number of event types.

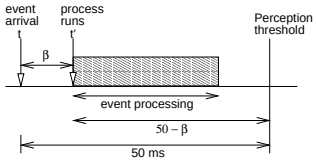


Figure 3: Event processing in a word processor

Instead we propose a different technique that can meet the human perception threshold without requiring explicit knowledge of various events types. Our technique referred to as *gradual processor acceleration (GPA)* accounts for the processor demand and the processor availability *implicitly*.

Upon the arrival of any event, the word processor is configured to run under at a low CPU speed, and a timer is set (the timer value is less that the perception threshold). If the processing of the event finishes before the timer expires, then the application simply waits for the next event. Otherwise, it increases the CPU speed by some amount and sets another timer. If the event processing continues beyond the timer expiration, the CPU speed is increased yet again and a new timer is set. Thus, the processor is gradually accelerated until either the event processing is complete or the maximum CPU speed is reached. In order to ensure adequate interactive performance, the maximum CPU speed is always used when the event processing time exceeds the perception threshold.

To understand how to instantiate this policy, suppose that the event arrives at time t and the application is actually scheduled on the CPU at time t' (although the application becomes runnable as soon as the event arrives, other concurrent applications can delay the scheduling of this application). From the perspective of the user, a response is desirable from the application no later than $t + 50$ ms. Since the application actually starts executing at time t', it needs to process the event within the remaining $50 - \beta$ ms, where $\beta = t' - t$ (see Figure 3). To do so, we choose n timers, which have values t_1, t_2, ..., t_n, and $\sum_{i=1}^{n} t_i = 50 - \beta$. After the expiration of the ith timer, the processor speed is increased to f_i, where f_i denotes a fraction of the maximum speed. The values of f_i are chosen such that the processor speed increases progressively and $f_n = f_{max} = 1$. Thus, the application runs at full processor speed if the event processing continues beyond $50 - \beta$ ms. Observe that, rather than explicitly estimating the processor demand of the event, the GPA technique monitors the progress of the event processing and adjusts the processor speed accordingly. Further, β implicitly captures the impact of other concurrent applications in the system.

Analysis: It is possible to bound the maximum slowdown incurred by an application in the GPA technique by carefully choosing timer values and CPU speeds. To see how, observe that if the processor were running at full speed, the amount of work done in the interval $[t', t' + \sum_{i=1}^{n} t_i]$ will take only $\sum_{i=1}^{n} f_i t_i$ at full processor speed. If the actual full-speed processing time of the event is smaller than this value the event finishes before the $50 - \beta$ ms perception threshold in the GPA technique, and thus the user does not perceive any performance degradation. For any event requiring more than this amount of full speed execution time, the maximum possible performance degradation under our strategy is given by:

$$degrade = 50 - \beta - \sum_{i=1}^{n} f_i \cdot t_i \quad (3)$$

since the processor will run at full speed once the execution time exceeds the perception threshold.

To illustrate, suppose that the maximum degradation is set to 20ms

over full processor speed. Let $\beta = 0$ for simplicity. If we choose five timers with values 30ms, 5ms, 5ms, 5ms, and 5ms, and the processor speeds during these timer intervals as 45%, 60%, 80%, 90%, and 100%, respectively, then, from Equation 3, the maximum possible user-perceived degradation for any event is 20ms. This is the maximum slowdown for any event requiring more than 50ms of processing time.

Implementation: We implemented GPA into AbiWord, a sophisticated word processor with a code base of hundreds of thousands lines of C code. Our implementation was straight-forward. We added code at the beginning of the AbiWord event handler to implement the GPA technique. The X11-server assigns a time-stamp to each new user event such as mouse click or key-stroke. We extracted this time-stamp t and used `gettimeofday` to determine the execution start time t'. The parameter β is computed as the difference between t' and t. This took only 17 lines of C code while setting timers and invoking the Chameleon interface took 23 lines of C code. In all, the implementation of GPA took only 40 lines of C code—a fairly modest change.

3.3 Web Browser

A web browser is another example of an event-driven interactive application that needs to process various events such as a mouse click or a keystroke. When the user types a URL or data into a web form, the keystrokes need to be displayed on the screen. When the mouse is positioned over a hyperlink, visual feedback needs to be provided by changing the shape of the mouse cursor. When the user clicks on a link, the browser needs to construct and send out a HTTP request; when data arrives from the remote server, it needs to parse and display the incoming data. Although the network delay is beyond the control of the browser, all other "local" events should be processed within the human perception threshold for good interactive performance. The GPA technique can be directly used for power management in such a browser.

We chose *Dillo*, a compact, portable open-source browser that runs on desktops, laptops and PDAs and implemented the GPA technique into this browser. Like in the case of the word processor, our modifications were restricted to the event handler in Dillo. First, we extracted the event arrival time and the execution start time in the event handler to compute β. We then added code to set timers and increase the processor speed upon timer expiration. In all, the implementation of GPA into Dillo involved 46 lines of C code, again demonstrating the modest nature of our modifications.

3.4 Batch Compilations

Compilations using *make* is an example of a batch application. Unlike interactive applications where the response time is important, the throughput is important for batch applications. Typically, *make* spawns a sequence of compilation tasks, one for each source code file. One possible user-level power management strategy is to estimate the processor demand for each compilation task and to choose an appropriate speed setting. However, since each compilation task is relatively short-lived, gathering CPU statistics for each process is difficult. Instead, a better strategy is to allow the end-user to specify the desired speed setting. System defaults can be used when the user does not specify a setting.

Most Unix-like operating systems support the *nice* utility, which allows the end-user to specify a CPU scheduling priority prior for a new process. For instance, the user can invoke the command `nice -n N make` to specify that *make* should run at priority N. A low priority enables the batch application to run in the background without interfering with foreground interactive applications. A high priority can also be chosen if the new application is more important than current applications.

A similar strategy can be used for choosing CPU speed settings. We implemented a utility called *pnice* that enables the end-user to specify a particular CPU speed setting for a new process. For instance, the user can invoke the command `pnice -n N make` to specify that make and all compilations spawned by its should run at a fixed CPU speed setting N. A lower speed setting enables energy savings at the expense of increasing the completion time, whereas a higher setting lowers the completion time at the expense of higher energy consumption.

Implementation of *pnice* was straightforward. The pnice process first changes its own speed setting to the specified value N using the Chameleon interface. Next, it invokes *exec* to run the specified command. This ensures that the application inherits the speed setting of the *pnice* process. The Chameleon kernel implementation ensures that any process *forked* by a parent process inherits the CPU speed setting of the parent. The *pnice* utility was implemented in 125 lines of C code, again demonstrating that implementation of user-level power management policies take modest effort.

4. USER-LEVEL POWER MANAGER

The previous section demonstrated how many commonly used applications can implement their own power management strategy. However, implementing a user-level power management strategy requires modification to the source code, which may not be feasible for legacy applications. Such applications can delegate the task of power management to a user-level power manager. The power manager can use CPU usage statistics and any application-supplied knowledge to modify CPU speed settings on behalf of the applications. A simple user-level power manager may choose a single speed setting for all applications based on current utilization; the speed setting is varied with observed changes in system utilization. A more complex strategy is to choose a different speed setting for each individual application based on its observed behavior; doing so requires usage statistics to be maintained for each application. Multiple user-level power managers can coexist in the system, so long as each manages a mutually exclusive subset of the applications. Thus, it is feasible to implement a different power manager for each class of application.

The Chameleon interface enables the entire range of these possibilities. To demonstrate the flexibility of our approach, we take a recently proposed DVFS approach—GraceOS [26, 27] —and show how it can be implemented as a user-level power manager using Chameleon. Our objective is two-fold. First, we show that many recently proposed approaches such as GraceOS that employ an *in-kernel* implementation can be implemented as user-level power managers in our approach. Second, GraceOS advocates a cooperative application-OS approach, where applications periodically supply information to the OS and the OS chooses the processor speed setting based on this information and usage statistics. We show that such interactions between the application and the CPU scheduler are feasible using Chameleon.

Implementation: We begin with a brief overview of the GraceOS [26, 27]. GraceOS is designed for periodic multimedia applications that belong to the soft real-time class. GraceOS treats such applications differently from traditional best-effort applications. Whereas best-effort applications are scheduled using the Linux time-sharing scheduler and do not benefit from DVFS, soft real-time applications are scheduled using a QoS-aware soft real-time scheduler and benefit from DVFS.

To handle soft real-time applications, GraceOS employs two key components: (i) a real-time scheduler and (ii) a DVFS algorithm. The CPU scheduler is vanilla earliest deadline first (EDF); standard EDF theory is used to perform admission control of soft real-

time tasks based on their worst case CPU demands. Admitted soft real-time tasks have strict priority over best-effort tasks. Deadlines derived from the application-specified periods are used for EDF scheduling of these tasks. Three system calls—*EnterSRT*, *ExitSRT*, and *FinishJob*—are used to convey start and finish time of tasks (e.g., frame decode) to the scheduler.

The DVFS algorithm maintains a histogram of CPU usage and derives a probability distribution of processor demand. The processor demand and the application-specified periods are used in a dynamic programming algorithm to derive a list of speed scaling points. Each point (x, y) specifies that a job should runs at the speed y when it has used x cycles. The DVFS algorithm monitors the cycle usage of the task. If the usage increases beyond x, the next speed setting y is chosen. Observe that this technique has similarities with our GPA technique where the progress of a task is monitored and the speed is increased gradually. The key difference is that the durations x and speeds y are computed at run-time using dynamic programming, whereas in GPA, they are statically chosen.

To implement GraceOS as a user-level power manager, we must distinguish between the DVFS component and the CPU scheduler. The DVFS algorithm is fully implemented in user space and uses the Chameleon interface to query usage statistics and monitor progress. The CPU scheduler and any interactions between the application and the scheduler must be implemented separately from Chameleon. Since Chameleon does not make any specific assumptions about the underlying scheduler, it is compatible with any CPU scheduling algorithm, including EDF.

Consequently, our implementation of the GraceOS includes three components: (i) a user-level daemon to calculate the soft real-time task's demand distribution, cycle budget, and speed schedule using dynamic programming (300 lines of C code); (ii) use of Chameleon's */dev/syscpu* interface driver to query the actual usage of each soft real-time task (109 lines of C code); and (iii) three system calls *EnterSRT*, *ExitSRT*, and *FinishJob* that allow an application to convey the beginning and end of each soft real-time task (23 lines of C code). Observe that the first two components relate to the DVFS algorithm, while the third component is used by the CPU scheduler in GraceOS. The GraceOS user-level power manager runs at the highest CPU priority in our system. All soft real-time applications run at the next highest CPU priority, and best effort jobs run at lower priorities. EDF scheduling is emulated by manipulating priorities of tasks; the task with the earliest deadline is elevated in priority (analogous to the implementation of EDF in GraceOS).

5. IMPLEMENTATION

Our prototype of Chameleon is implemented as a set of modules and patches in the Linux kernel 2.4.20-9.

New system calls: We added four new system calls to implement the Chameleon OS interface: (i) *get-speed*, which returns the current CPU speed of the specified process or process group; (ii) *set-speed*, which sets the CPU speed of the specified process or process group; (iii) *get-speed-schedule*, which returns processor budget and speed schedule of the specified process, and (iv) *set-speed-schedule*, which sets the processor budget and speed schedule of the specified task. The latter two system calls enable sophisticated speed setting strategies, where an application can specify an *a priori* schedule for changing the speed as it executes.

Chameleon-enhanced /proc interface: We enhanced the */proc* interface by adding a /proc/Chameleon sub-tree. This directory holds one file for each Chameleon-driven process and allows applications to query their CPU quantum allocations in the recent past.

Chameleon /dev interfaces: To support user-level power managers, we added two new /dev interfaces: */dev/sysdvfs* and */dev/syscpu*. The system-wide utilization is reported via */dev/sysdvfs*, whereas the CPU cycles consumed by individual tasks are reported via */dev/syscpu*.

Process control block enhancements: In order to allow Chameleon to implement techniques such as PACE [14, 15] and GraceOS [26, 27] as user-level power managers, we borrowed several process control block attributes from the GraceOS implementation: (i) cycle counter, which measures the CPU cycles used by a task, (ii) cycle budget, which stores the number of allocated cycles, and (iii) speed schedule, which stores a list and schedule of speed scaling points. Whereas these three attributes are meaningful only for Chameleon processes managed by user-level power managers, we also added three more attributes that are applicable to all processes in the system: (i) Chameleon-driven-flag, which indicates if the process is directly modifying its speed settings; (ii) current-speed, which specifies the current CPU speed setting of the process; (iii) inheritable-flag, which indicates if the speed setting is inheritable by its children.

DVS kernel module: The DVS kernel module is actually responsible for interfacing with the hardware in order to modify the processor speed. This is done by writing the frequency and voltage to two machine special registers (MSR) [26, 27]. Chameleon can be applied to any DVFS-enabled processor by implementing a DVS kernel module specific to that processor.

Linux scheduler enhancements: We modified the standard scheduler to add per-process speed settings and cycle charging. Similar to our process control block enhancements, cycle charging is only necessary to implement other techniques as user-level power managers, and is directly inspired by the GraceOS implementation [26, 27]. Whenever the schedule() function is invoked, the modified scheduler will do the following: (i) in the case of no context switch, it may change the speed of the current task according to its speed schedule; (ii) in the case of a context switch, the scheduler performs some book-keeping only for the previous task with a speed schedule (e.g., update its cycle counter, decrement cycle budget, advance speed schedule, etc.); (iii) then the scheduler sets the CPU speed for the new task based on its current-speed attribute.

Our implementation of Chameleon runs on a Sony Vaio PCG-V1CPK laptop with Transmeta Crusoe TM5600-667 processor [23]. The Transmeta TM5600 processor supports five discrete frequency and voltage levels (see Table 1) and implements the *LongRun* [7] technology in hardware to dynamically vary the CPU frequency based on the observed system-wide CPU utilization. LongRun varies the CPU frequency between a user-specified maximum and minimum values—these values can be set by users by writing to two machine special registers (MSR). By default, these values are set to 300 MHZ and 677 MHz, enabling the full range of voltage scaling. LongRun can be disabled by setting the minimum and maximum register values to the same frequency (e.g., setting both to 533 MHz does not allow any leeway in changing the CPU frequency, effectively disabling LongRun). This feature can be used to implement voltage scaling in *software*—the power-aware application can determine the desired frequency and set the two registers to this value.

Freq. (MHz)	Voltage (V)	Power (W)
300	1.2	1.30
400	1.225	1.90
533	1.35	3.00
600	1.5	4.20
667	1.6	5.30

Table 1: Characteristics of the TM5600-667 processor

6. EXPERIMENTAL EVALUATION

We evaluated Chameleon on a Sony PCG-V1CPK laptop equipped with a Transmeta Crusoe processor and 128MB RAM. The operating system was Red Hat Linux 9.0 with a modified version of kernel 2.4.20-9. To compare Chameleon with other DVFS approaches, we implemented three OS-based DVFS techniques proposed in the literature: (i) PAST [24], (ii) PEAK [12], and (iii) AVG_n [11], all of which are interval-based system-wide DVFS techniques. Our experiments involve running applications under six different configurations: (i) with DVFS disabled—the CPU always runs at the maximum speed (denoted as FULL), (ii) using the hardwired LongRun technology, (iii) using PAST, (iv) using PEAK, (v) using AVG_n, and (vi) using Chameleon (where LongRun is disabled for power-aware applications but enabled for legacy applications).

The energy consumption of the processor during an interval T is computed as

$$energy = \sum_{i=1}^{n} p_i t_i \quad (4)$$

where n is the number of available frequency/voltage combinations on the processor, p_i denotes the power consumption of the processor when running at the ith frequency/voltage combination, and t_i represents the time spent at the ith frequency/voltage combination during the interval T. We modified the Linux kernel to record the energy consumption of the TM5600 processor using Equation 4 and Table 1. Given the energy consumption of the processor during an interval T, the average power consumption of the processor during this interval is computed as $power_{avg} = \frac{energy}{T}$. Our experiments showed that PEAK always consumed the least processor energy among all DVFS techniques. However, it trades its energy savings with an unacceptably high performance degradation for time-sensitive multimedia and interactive applications. For example, video decoding of a 30 minutes clip took an extra 16.6 minutes, resulting in poor performance. Therefore, we omit the results of PEAK in the rest of this paper and refer the readers to [13] for these results.

6.1 Chameleon-aware Applications

We first evaluate our four Chameleon-aware applications. Our experiments assume a lightly-loaded system that runs a single application with the typical background system processes.

6.1.1 Video Decoder

We encoded several DVD movies at different bit-rates and resolutions using Divx MPEG2/MPEG4 video codec and MP3 audio codec The characteristics of six such movies are listed in Table 2. The bit-rates are depicted in the form $(a + b)$Kbps, where a is the video and b is the audio bit-rate. We recorded the energy consumed by the processor during playback of these movies at full speed, with LongRun, with Chameleon, with PAST, and with AVG_n.

	Res.	Length	Frames	Bit-Rate(Kbps)
Movie 1	640x272	3360s	80387	1290.9 + 179.2
Movie 2	640x272	612s	14577	757.2 + 128.0
Movie 3	640x352	7168s	179168	679.7 + 128.0
Movie 4	640x352	602s	15003	861.9 + 128.0
Movie 5	640x352	1755s	42040	2456.9 + 192.0
Movie 6	640x480	2394s	57355	1674.6 + 384.0

Table 2: Characteristics of MPEG 4 Videos

Our experiments show that all five configurations handle movie playback very well. The same playback quality is observed under: identical execution times which equal the length of the movies, identical frame rates, no dropped frames, and no user-noticeable delays. However, the average CPU power consumption differs significantly across the various configurations (see Figure 4(a)). Figure 4(a) shows that: (i) neither PAST nor AVG_n can outperform LongRun; (ii) LongRun can achieve significant energy savings (from 27.36% to 57.26%) when compared to FULL; (iii) the Chameleon-aware *mplayer* can achieve an additional 20.52% to 31.99% energy savings when compared to LongRun.

Although there are no user-perceived playback problems (in terms of dropped frames or playback freezes) under the five configurations, we do observe jitter in the playback quality at the frame-level. Such small inter-frame jitter is inevitable in a time-sharing CPU scheduler, although its effects are not perceptible at the user-level. *mplayer* provides statistical measurements of late frames—the number of frames that are behind their deadline by more than 20% of the frame interval. As shown in Figure 4(b), the number of late frames in Chameleon is mostly comparable to PAST and AVG_n and typically better than LongRun (while consuming the least energy). FULL has the least—although not zero—late frames at the expense of the highest energy consumption. The number of late frames is small $(0.2 - 2.3\%)$ in all configurations.

6.1.2 Web Browser and Word Processor

We ran the web browser and the word processor and measured their average power consumption, the average response time, and the percentage of late events (where event processing time exceeds the 50ms threshold).

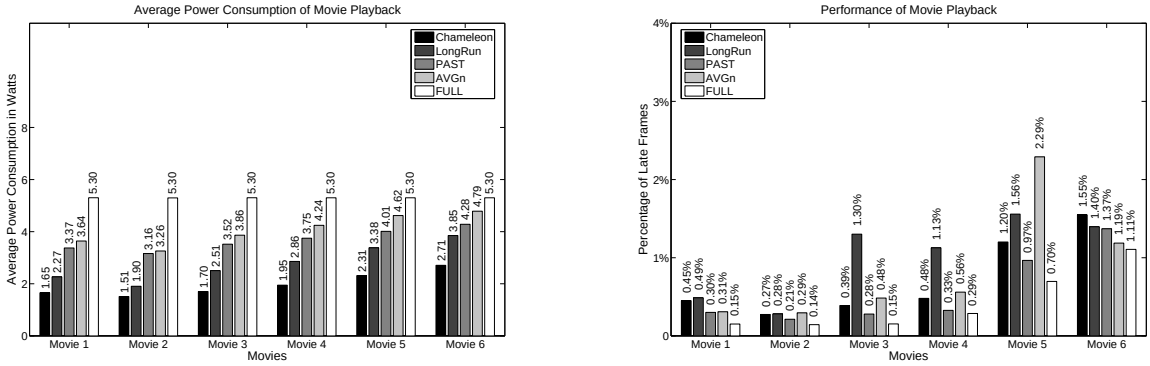
To eliminate the impact of variable network delays, our experiments with the web browser consisted of a client requesting a sequence of web pages from a web server on a local area network; the requested web pages consist of actual web content that was saved from a variety of popular web sites. Each experiment consists of a sequence of requests to these web pages with a uniformly distributed "think-time" between successive requests. The experiments differ in the requested web pages and the chosen think times; each experiment is repeated under the five configurations, and we measure the mean for each experiment.

The workload for the word processor emulates a user editing a sequence of documents. Each experiment contains a script that makes a sequence of editing requests to these documents with a uniformly distributed "think-time" between successive requests. The experiments differ in the edited documents and the chosen think times; each experiment is repeated under the five configurations, and we measure the mean for each experiment.

Figure 5(a) shows that LongRun consumes a factor of three less power than FULL. Chameleon is able to extract an additional 10.27% energy savings when compared to LongRun, while PAST is worse than LongRun. We also note that the average power consumption under Chameleon is only 0.03W and 0.06W higher than the power consumption at the slowest CPU speed (300MHz) for the browser and the word processor, respectively. Further, most events finish in Chameleon without any performance degradation. The percentage of late events is only 0.24% and 0.22% in the word processor and the browser, respectively, and is comparable to other approaches. Finally, the increase in processing times of late events is no more than 20ms (obtained by substituting the chosen timer values and CPU speeds in Equation 3).

6.1.3 Batch Compilations

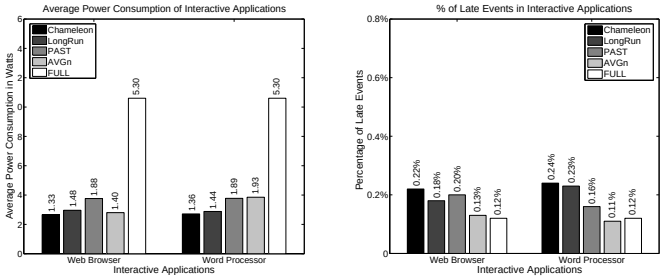
We compiled a portion of the *ns-2* network simulator using *make* and our *pnice* utility. We chose different values of the CPU speed in *pnice* and measured the power consumption and completion times of *make*. As expected, our results, depicted in Table 3, show that the power consumption can be traded for completion time by appropriately choosing a speed setting. Faster speeds lower completion



(a) Average CPU Power Consumption (Watts)

(b) % of Late Frames

Figure 4: Average CPU power consumption and percentage of frames that are late by more than 8ms (20% of the 40ms deadline).



(a) Average Power Consumption (Watts)

(b) % of Late Events

Figure 5: Average CPU power consumption and % of late events.

times at the expense of higher energy.

Freq. (MHz)	Completion Time	Mean Power Consumed
300	1376s	1.38W
400	1066s	1.96W
533	910s	3.00W
600	812s	4.14W
667	776s	5.15W

Table 3: Completion times and mean CPU power consumption for batch compilations.

6.2 Isolation in Chameleon

We claim that Chameleon isolates an application from the power settings of other applications. To demonstrate this isolation, we ran *mplayer* with a misbehaving background application. The background application rapidly switches its CPU speeds from one setting to another every few milliseconds. We ran mplayer with this application when it was well-behaved (it used a fixed CPU speed throughout) and then with the misbehaving version of the application. We measured its impact on the progress of the mplayer. Our results show that the progress made by mplayer is unaffected by the rapid changes of CPU speed by the misbehaving application—any change to the CPU speed by an application only impacts its own progress and has no impact on the CPU shares of other applications.

6.3 Impact of Concurrent Workloads

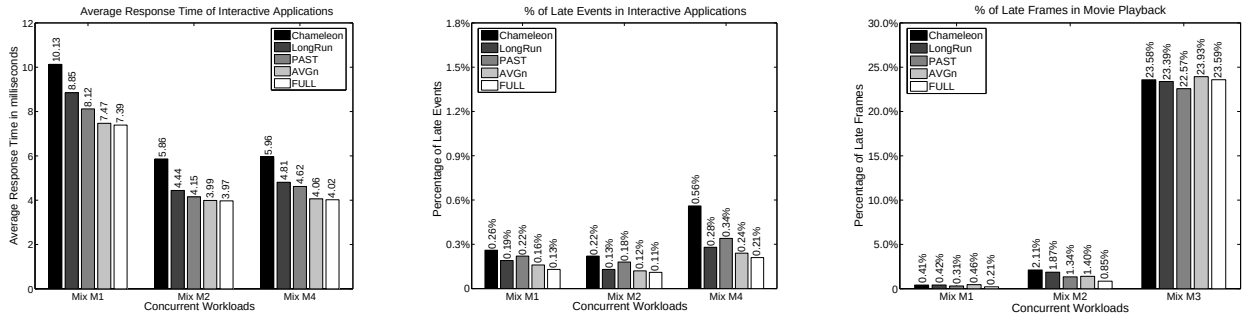
To demonstrate that applications can make locally- and globally-optimal power management decisions in the presence of concurrent applications, we considered four application mixes: (i) video decoder and web browser (mix M1), (ii) video decoder and word processor (mix M2), (iii) video decoder and batch compilations (mix M3), and (iv) batch compilations and word processor (mix M4). Note that, from the perspective of the video decoder, the background load increases progressively from mix M1 to M3.

Table 4 and Figure 6 show the average power consumption and the performance of these applications under various power management strategies. Table 4 shows that Chameleon always consumes the least energy among the five configurations. The energy savings range from 19.81% to 31.19% when compared to LongRun, which itself extracts up to 41.89% reduction when compared to FULL. The performance degradation, depicted in Figure 6(a), shows that interactive application performance in Chameleon is comparable to the other techniques. For instance, the average event processing time of the word processor under mix M2 increases from 4.4ms in LongRun to 5.96ms in Chameleon and is well under the human perception threshold of 50ms. A similar result is seen for the web browser under mix M1. The percentage of late events remains well under 1% under all mixes (see Figure 6(b)).

Figure 6(c) plots the percentage of late frames in the video decoder for different mixes. The figure shows that the percentage of late frames in Chameleon is comparable to other approaches. As the background load increases from mix 1 to mix 3, we see that the percentage of late frames increases from around 0.4% to more than 22%. For mix M3, all techniques, including FULL, incur 22% deadline misses. Decoding of the 10 minute clip takes an extra 20 seconds under all techniques, resulting in poor performance. This is primarily due to insufficient processor availability at higher loads, as opposed to deficiencies in the power management technique. Due to the background load imposed by the batch compilations in mix M3, the time sharing scheduler is unable to allocate sufficient CPU time to the video decoder.

Figure 7 shows the fraction of time spent by the video decoder at different CPU speed settings. In the absence of any background load, the decoder is able to lower its speed setting to the lowest speed for more than 90% of the time. As the load increases, the fraction of time spent at higher speeds increases. For mix M3, more than 80% of the time is spent at the highest speed (recall that insufficient processor availability causes the video decoder to run at full speed—Case 2 in Section 3.1).

Under mix M3, the only possible solution is to use a QoS-aware



(a) Average Response Time (Milliseconds) (b) % of Late Events (c) % of Late Frames

Figure 6: Performance of concurrent applications: average response time of interactive applications and % of late events and frames.

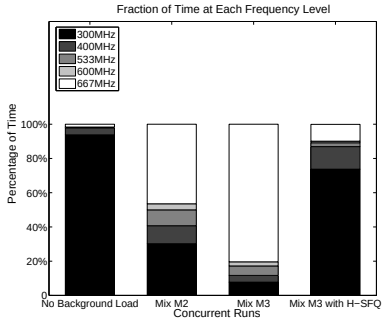


Figure 7: Fraction of time at various frequency levels by *mplayer*

scheduler that guarantees a fixed fraction of the CPU to the video decoder regardless of the background load. We ran mix M3 with Chameleon on a proportional-share scheduler, namely Hierarchical Start Time Fair Queue (HSFQ) CPU scheduler [10]. In this experiment, we assigned $1/14$ fraction of CPU time to the batch compilations, $12/14$ fraction of CPU time to the video decoder and the X server, and the remaining $1/14$ to the other tasks. As expected, the percentage of late frames in the video decoder fell to a very small value. Further, since processor availability is guaranteed in HSFQ, as shown in Figure 7, the video decoder was able to spend 73.73% of its execution time at the lowest frequency (300MHz) (compared to 7.74% under time-sharing CPU scheduler). This causes the mean power consumption to fall to 2.1W, a 44.8% reduction when compared to the time-sharing scheduler.

	Chameleon	LongRun	PAST	AVG_n	FULL
Mix M1	2.25W	3.27W	3.98W	4.42W	5.3W
Mix M2	2.47W	3.08W	3.79W	3.83W	5.3W
Mix M3	3.81W	5.27W	5.26W	5.27W	5.3W
Mix M4	3.71W	5.22W	5.23W	5.23W	5.3W

Table 4: Average CPU Power Consumption for various mixes.

6.4 User-level Power Manager Experiments

We modified *mplayer* to use the GraceOS system calls and used it to decode the movies in Table 2. The GraceOS user-level power manager was used to make power management decisions on behalf of mplayer. We measure the energy consumed by mplayer and plot it in Table 5. Our results show that GraceOS can achieve 3.50% to 18.44% energy savings when compared to LongRun. However, Chameleon outperforms GraceOS by 9-41%. This is because the Chameleon-enhanced mplayer is able to estimate decode times of individual frames based on domain-knowledge, while GraceOS relies on external observations of the CPU usage of mplayer. This domain knowledge yields an extra 9-41% in Chameleon.

Movies	AVG. Power	Eng. Sav. to LongRun	to Chameleon
Movie 1	2.11W	7.05%	-27.88%
Movie 2	1.64W	13.68%	-9.33%
Movie 3	2.11W	15.94%	-24.12%
Movie 4	2.76W	3.50%	-41.54%
Movie 5	3.09W	8.58%	-33.77%
Movie 6	3.14W	18.44%	-13.69%

Table 5: GraceOS CPU power consumption for movie playback

6.5 Implementation Overhead

An important consideration is the overhead caused by frequent changes in the CPU speed setting. Using the CPU cycle counter register, we measure the cost as 1125 cycles (about 3.75 μs under 300 MHz and 1.69 μs under 667 MHz). Due to better DVFS support in the Transmeta processor, this is considerably lower than the 8,000-16,000 cycles reported for an HP laptop used in the GraceOS experiments [26, 27]; however, both incur minimal overhead. Finally, the overhead values of the video decoder, GPA and *pnice* strategies are 2738 per frame, 1149 per timer, and 127 CPU cycles, respectively, which is in the order of a few micro-seconds.

7. RELATED WORK

Recently, power management techniques for mobile devices have received increasing research attention. The proposed techniques either use dynamic voltage and frequency scaling (DVFS) [3, 16, 17, 19, 26, 27] or application/middleware-based adaptation [8, 9, 21, 22] for energy savings. DVFS approaches extract energy savings by varying the processor speed; the techniques do not affect the amount of processing performed by the application—the processing is merely spread over longer time periods by lowering CPU speeds. In contrast, middleware-based adaptation approaches vary quality or data fidelity and thus, the amount of processing performed by the application to extract energy savings. We review related work in both categories.

Application or middleware-based adaptation techniques trade the computational overhead for application quality; energy savings are extracted by reducing video quality [21, 22], document quality [8] or data fidelity [9], and thus, the processing overheads. Proxy-based adaptation for reducing video quality has been explored in [21, 22]. Puppeteer adapts document quality (i.e. picture resolution, color depth) for energy savings of office applications [4, 8]. The impact of adapting the data fidelity on energy savings of several applications has also been demonstrated in Odyssey [9].

In contrast, DVFS techniques do not reduce the amount of processing overhead imposed by an application; instead they vary the CPU speed to match the CPU load and extract energy savings [3,

16, 17, 19, 26, 27]. DVFS techniques fall into four categories: hardware-based, OS-based, cooperative application-OS-based, and application-directed methods. Hardware-based approaches such as LongRun [7] measure system utilization in hardware and choose a system-wide speed setting based on the current utilization. An online hardware approach for voltage and frequency control in multiple clock domain microprocessors has been proposed in [25]. OS-based approaches determine a system-wide CPU setting based on the processor demands of the currently active tasks [6, 14, 15, 20]. In this approach, individual applications do not have any direct control over the CPU power settings. A single system-wide CPU setting is determined, typically based on the needs of the most resource-hungry application, even when a mix of applications is executing on the processor. Furthermore, the OS needs to *infer* the processing needs of applications and can incur measurement errors. In cooperative application-OS approaches, the application provides some domain-specific information to the kernel. The OS kernel and the CPU scheduler use this information for CPU speed setting and/or scheduling. GRACE-OS [26, 27] proposes such a cooperative application/OS approach for periodic multimedia applications. It uses probability distributions of CPU usage of periodic applications and knowledge of application periods (which is supplied by the application) for choosing CPU speeds. Aperiodic or non-real-time applications are currently not handled by the approach.

An cooperative power management approach was proposed in [18] to unify low level architectural optimizations (CPU, memory, register), OS power-saving mechanisms (DVFS) and adaptive middle techniques (admission control, optimal transcoding, network traffic regulation). In this technique, interaction parameters between the different levels are identified and optimized to significantly reduce power consumption.

Rather than an OS-application partnership, Chameleon exports the entire burden of power management to the user level.

Finally, several different application-controlled DVFS techniques for video decoding have been proposed [3, 16, 17, 19]. While some require offline estimation of CPU demands for decoding [17], others can estimate the CPU demands online [3, 16, 19]. However, all of these techniques implicitly assume only a single application is executing on the CPU and grant complete control of the processor settings to the video decoder. Unlike in Chameleon, other applications are not considered—the issue of concurrent applications that might use a different speed setting is not considered in these efforts, nor is the issue of providing isolation across applications considered explicitly.

8. CONCLUSIONS

This paper argued that applications know best what their energy needs are and proposed Chameleon, an approach that puts the entire burden of power management on individual applications. Our implementation and experiments showed that (i) user-level policies can be implemented at a modest cost of tens of lines of code, (ii) Chameleon can extract up to 32% energy savings when compared to LongRun and up to 50% when compared to recent OS-based DVFS techniques, (iii) concurrent applications benefit from Chameleon's flexibility, and (iv) Chameleon imposes negligible overheads.

Acknowledgments: We would like to thank our shepherd Sid Ahuja and the anonymous reviewers for their comments. This research was supported in part by NSF grants CCR-0219520, EIA-0080119, CNS-0520729, CNS-0519881, CNS-0447877 and DUE-0416863.

9. REFERENCES

[1] A. Bavier, A. Montz, and L. Peterson. Predicting mpeg execution times. In *Proc. of ACM Sigmetrics'98*, pages 131–140, June 1998.

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

[3] K. Choi, K. Dantu, W. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proc. of the IEEE/ACM CAD'02, San Jose, CA*, pages 732–737, November 2002.

[4] E. de Lara, D. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proc. of USITS'01, San Francisco, CA*, pages 159–170, March 2001.

[5] D. R. Engler, M. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of ACM SOSP'95, Copper Mountain, CO*, pages 251–266, December 1995.

[6] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proc. of OSDI'02, Boston, MA*, pages 105–116, December 2002.

[7] M. Fleischmann. Longrun power management - dynamic power management for crusoe processors. Technical report, Transmeta Corporation, 2001.

[8] J. Flinn, E. de Lara, M. Satyanarayanan, D. Wallach, and W. Zwaenepoel. Reducing the energy usage of offi ce applications. In *Proc. of Middleware'01, Heidelberg, Germany*, November 2001.

[9] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of ACM SOSP'99, Charleston, SC*, pages 48–63, December 1999.

[10] P. Goyal, X. Guo, and H. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proc. of OSDI'96, Seattle, WA*, pages 107–122, October 1996.

[11] D. Grunwald, P. Levis, K. Farkas, C. M. III, and M. Neufald. Policies for dynamic clock scheduling. In *Proc. of OSDI'00, San Diego, CA*, pages 73–86, October 2000.

[12] K.Govil, E.Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proc. of ACM Mobicom'95, Berkeley, CA*, pages 13–25, November 1995.

[13] X. Liu, P. Shenoy, and M. Corner. Chameleon: Application controlled power management with performance isolation. Technical report 04-26, University of Massachusetts, 2004.

[14] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE In *Proc. of ACM Sigmetrics'01, Cambridge, MA*, pages 50–61, June 2001.

[15] J. R. Lorch and A. J. Smith. Operating system modifi cations for task-based speed and voltage scheduling. In *Proc. of ACM MobiSys'03, San Francisco, CA*, pages 215–229, May 2003.

[16] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proc. of ACM/IEEE CASE'01, Greenoble, France*, pages 156–163, October 2002.

[17] M. Mesarina and Y. Turner. Reduced energy decoding of mpeg streams. In *Proc. of MMCN'02*, pages 73–84, January 2002.

[18] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian. Integrated power management for video streaming to mobile handheld devices. In *Proc. of ACM MM'03, Berkeley, CA*, pages 582–591, November 2003.

[19] J. Pouwelse, K. Langendoen, I. Lagendijk, and H. Sips. Power-aware video decoding. In *Proc. of PCS'01, Seoul, Korea*, pages 303–306, April 2001.

[20] J. Pouwelse, K. Langendoen, and H. Sips. Application-directed voltage scaling. *IEEE Trans. on VLSI*, 11(5):812–826, October 2003.

[21] P. Shenoy and P. Radkov. Proxy-assisted power-friendly streaming to mobile devices. In *Proc. of MMCN'03, Santa Clara, CA*, pages 177–191, Janauary 2003.

[22] M. Tamai, T. Sun, K. Yasumoto, N. Shibata, and M. Ito. Energy-aware video streaming with qos control for portable computing devices. In *Proc. of ACM NOSSDAV'04, Cork, Ireland*, pages 68–73, June 2004.

[23] Crosoe tm5600 processor data sheet. Transmeta Inc., http://www.transmeta.com.

[24] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proc. of OSDI'94, Monterey, CA*, pages 13–23, November 1994.

[25] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proc. of ACM ASPLOS-XI, Boston, MA* , October 2004.

[26] W. Yuan and K. Nahrstedt. Energy-effi cient soft real-time cpu scheduling for mobile multimedia systems. In *Proc. of ACM SOSP'03, Bolton Landing, NY*, pages 149–163, October 2003.

[27] W. Yuan and K. Nahrstedt. Practical Voltage Scaling for Mobile Multimedia Devices. In *Proc. of ACM MM'04, New York, NY*, pages 924–931, October 2004.