Operating System Support for CPU Power Management in Mobile Devices

Y. Hassan U. Khan M. Naji K. Salah
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
Email: {yshassan, omikhan, mohnaji,salah}@ccse.kfupm.edu.sa

Abstract

Dynamic Power Management (DPM) is a technique to dynamically adjust mobile devices for optimum power consumption. One of the ways of implementing it is by dynamically scaling the voltage supply and in turn varying the clock cycle, and thus resulting in changing the energy-state of the device. In this paper we qualitatively compare the most popular voltage scaling systems that exist in literature. We first survey and describe the various systems for voltage scaling and then present various criteria to analyze them. These criteria include energy state classification, task scheduling algorithm, task CPU usage, and CPU speed scaling policy. This paper yields insight into analyzing the various parameters that influence the composition of these systems. We also propose extensions to existing ideas to enhance the flexibility and optimality of existing DVS systems.

KEYWORDS: Mobile Devices, Operating Systems, Power Management, Scheduling Algorithms

1. Introduction

Mobile computing has come a long way since its start. The primary challenge that was faced lied in designing the various soft-wares that would function while prioritizing the three essential features of mobile computing, namely: communication, mobility and portability, as discussed in [1]. Research prospects were identified in areas such as cashing metrics, semantic callbacks and validators, resource revocation, adaptation and global estimation based on local observations [2]. Others classified these areas as mobility, scalability, bandwidth and energy management [3]. Optimizing energy usage in mobile systems, which are running on local battery, is of great importance [4]. Optimal energy management can be done with respect to several components including the hard disk, communication device, display device, the processor etc. Prior research has proved that it is the processor that consumes the maximum power. Thus, temporarily putting the CPU in low-power state can save energy [5]. Hence, the primary objective is to design a low-power-energy consumption technique.

The techniques to achieve power management can be broadly classified as *static* and *dynamic* [6]. Static techniques are applied at design time whereas dynamic techniques are applied at run-time. Dynamic techniques

use the run-time behavior of the system to reduce power. Managing power using the dynamic techniques is known as *dynamic power management* (DPM). DPM can be done in various ways. These include frequency reduction, voltage scaling, capacitance reduction or reducing switching activity [7]. These techniques are primarily used to reduce the power dissipation. The average power dissipation can be described by the following equation:

$$P_{avg} = P_{dynamic} + P_{short} + P_{leakage} + P_{static} \quad (1)$$

where, $P_{dynamic}$ is the dynamic power consumption, P_{short} is the short-circuit power consumption, $P_{leakage}$ is the leakage consumption and P_{static} is the static power consumption.

The most dominant factor is the dynamic power consumption. Hence, considering T denotes the clock period, C_{out} denotes the output capacitance, V_{dd} denotes the power supply, K is the average number of transitions in a clock cycle and f is the clock frequency; the following formula for average dynamic power consumption can be formulated [7].

$$P_{dynamic} = K\frac{C_{out}V_{dd}^2}{T} = KC_{out}V_{dd}^2 f \quad (2)$$

Of the various variables present in the above equation; only V_{dd}^2 can be altered at run-time. The other variables are fixed at design time. Thus they are related to static power management. Changing V_{dd}^2 is known as *Dynamic Voltage Scaling* (DVS). When the voltage is changed during run-time, the clock cycles of the device is reduced. This results in reduced heat dissipation. In effect; the device is switched to a lower energy-consumption state. From this we deduce that scaling the voltage results in a change of state of the device.

Changing the state of the device can be implemented in various ways. These implementations have been discussed in the following section along-with the algorithms that will be compared. The rest of the paper is organized as follows. Section 2 introduces the concept of DVS and the components that constitute a complete DVS system. It also summarizes the systems that will be compared. Section 3 proposes and discusses the comparison criteria used to compare the algorithms. Section 4 compares the DVS systems based on the criteria developed in Section 3. Finally, Section 5 concludes the comparison and identifies areas for future work.

2. Dynamic Voltage Scaling (DVS)

Dynamic Voltage Scaling (DVS) is one of the most effective techniques to save energy in mobile devices. DVS implies a set of techniques that can adjust the clock speed of the processor at the run time depending on the nature of task executing and doing this without missing the job performance deadline.

This takes advantage of the fact that in CMOS logic, the energy required per task is directly proportional to the product of speed and voltage squared. This can be mathematically expressed as follows:

$$\frac{Energy}{sec} \propto Voltage^2 \times speed \tag{3}$$

The term $speed$ can be expressed as a relation of $task$ and sec. Replacing this value gives us the following relation.

$$\frac{Energy}{sec} \propto Voltage^2 \times \frac{task}{sec} \tag{4}$$

or,
$$\frac{Energy}{task} \propto Voltage^2 \tag{5}$$

If assumed that voltage may be decreased in direct proportion to speed [4]; we can substitute speed in place of voltage in equation (5). This gives us equation (6).

$$\frac{Energy}{task} \propto Speed^2 \tag{6}$$

It is also evident from equation (5) that $Energy$ is quadratically proportional to $Voltage$. The above relation explains that if we reduce speed, it will reduce energy consumption quadratically. Thus,

$$Energy \propto Voltage^2 \tag{7}$$

When peak performance is not needed, the frequency and voltage can be reduced, and hence energy dissipation is also greatly reduced. Since reducing the voltage results in the quadratic decrease in the energy consumption, that's why if an effective way to predict workload is found then it would be better to spread the task by reducing cycle time instead of running the CPU at full speed for short bursts and then sitting idle [4]. This is described in Figure 1 [8]. It shows different runs of the same workload; comparing them on performance and time parameters. In A, the CPU functions at full throttle and hence finishes the task well in advance of the deadline and thus wastes the remainder of the time. However, in B, the task is stretched to its deadline. This reduces the performance but allows for energy savings as the voltage scaling has been applied to the processors.

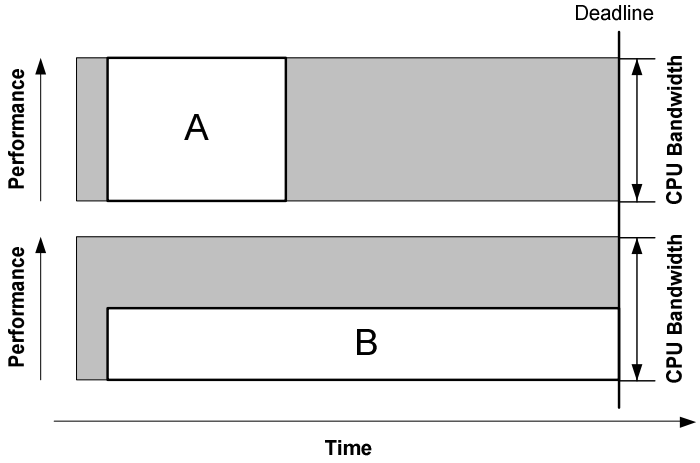

Figure 1. Performance scaling

Top

Dynamic Voltage reduction is not possible without scheduling algorithm that could determine the operating speed of the processor at run time. The *scheduler* needs a *profiler* that could tell how many clock cycles each job would need. The performance and specially responsiveness of a DVS algorithm also depend upon how well it estimates and utilizes the idle time. This is also called as '*Slack time analysis.*' But for mobile systems, which are usually hard or soft real time systems, this analysis can only be afforded when it is composed of some simple heuristics otherwise doing this analysis for every job will be a big overhead. In order to implement DVS, a speed setting policy is needed that would determine how to adjust the task execution in the coming clock cycle and a scheduling algorithm that schedules real time tasks so that there is minimum slack time and minimum missed deadlines.

2.1 Existing DVS Systems

This section discusses five of the various existing DVS systems. These systems are GRACE-OS, Real Time Hopping Algorithm (RVH), Low Power Fixed Priority Scheduling (LPFPS), Episode Based Algorithm and Low-Power scheduling using Slack Estimation Heuristic (lp/SEH.) These systems will then be compared based on criteria defined in the next section.

GRACE-OS. GRACE-OS [9] is a system that uses the "Application runtime CPU usage" strategy to predict the CPU demands of different applications. This in integrated with SRT scheduling to support quality-of-service. The combination of SRT scheduler and DVS algorithms result in an enhanced scheduler (GRACE-OS) which decides how fast to execute applications in addition to how long to execute them. Thus, the system supports dynamic time-quantum policies. The system employees a stochastic approach to allocate cycles based on the statistical performance requirements and probability distribution of cycle demands of individual application tasks. The system also supports task-level energy saving mechanism by implementing a speed controller for each task based on the previous cycle demands of the process. This enables each task to start slowly and then accelerate as the job continues. The system reduces CPU idle time and spends more busy time in low-power speeds.

There are three main components of GRACE-OS. They are the profiler, SRT scheduler and the speed adaptor. The profiler monitors the cycle usage of individual tasks and automatically derives the probability distribution of their cycle demands from the cycle usage. The SRT scheduler is responsible for allocating cycles to tasks and scheduling them to deliver performance guarantees. It performs soft real-time scheduling based on the statistical performance requirements and demand distribution of each task. Scheduling is dependent on the prediction of task cycle demands. This is a two step process which includes profiling the cycle usage and deriving the probability distribution of usage. The speed adaptor adjusts CPU speed dynamically to save energy. It adapts each task's execution speed based on the task's time allocation, provided by the SRT scheduler, and demand distribution, provided by the profiler. A cycle counter is added into the process control block of each task. As a task executes, its cycle counter monitors the number of cycles the task consumes. In particular, this counter measures the number of cycles elapsed between the task's switch-in and switch-out in context switches. The sum of these elapsed cycles during a job execution gives the number of cycles the job

uses. A speed schedule for each task is defined. The speed schedule is a list of scaling points. Each point (x, y) specifies that a job accelerates to the speed y when it uses x cycles. Among points in the list, the larger the cycle number x is, the higher the speed y becomes. The point list is sorted by the ascending order of the cycle number x (and hence speed y). According to this speed schedule, a task always starts a job at the speed of the first scaling point. As the job is executed, the scheduler monitors its cycle usage. If the cycle usage of a job is greater than or equal to the cycle number of the next scaling point, its execution is accelerated to the speed of the next scaling point. The speed schedule construction, for each task is to find a speed for each of its allocated cycles, such that the total energy consumption of these allocated cycles is minimized while their total execution time is not more than the allocated time.

Real Time Voltage Hopping Algorithm. Real Time Voltage Hopping Algorithm (RVH) [10] is a run-time dynamic voltage scaling scheme for low-power real-time systems. It employs software feedback control of supply voltage, which is applicable to off-the shelf processors. It provides efficient power reduction by fully exploiting slack time arising from workload variation. Using software analysis environment, the proposed scheme is shown to achieve 80~94% power reduction for typical real-time multimedia applications.

In real time systems, the utilization of the processor is frequently less than 1 even if all the tasks are executing at their worst-case execution time (WCET), implying that there is always some slack time. This slack time is exploited to lower the supply voltage. Most of the approaches only exploit the worst case slack time since they assume the task to be running at its WCET. However, this approach cannot fully take advantage of workload variation slack time because it controls voltage supply on task by task basis. Moreover in DVS systems, system clock frequency can have arbitrary values, which may cause interface problems to exchange data. Especially, this interface problem becomes serious for peripherals or other systems at different clock frequency

Real Time Voltage Hopping algorithm remedies this problem by utilizing the workload slack time of the tasks. This algorithm has the following features (1) relationship between clock frequency and supply voltage is measured by experiment and is stored as a look up table in the device driver (2) it controls clock frequency and supply voltage by software feedback which can be easily adopted for various targets (3) it avoids interface problems by exploiting discrete levels of clock frequency as f_{clk}, $f_{clk}/2$, $f_{clk}/3$……where f_{clk} is the highest clock frequency and (4) it fully utilizes workload-variation slack time by partitioning a task into several pieces, which we call time slots then dynamically controlling supply voltage on timeslot by timeslot basis.

Voltage scaling is done by the device driver. It has two lookup tables: one for voltage-frequency relationship of the target processor, and the other for transition delay to change clock frequency and supply voltage. These lookup tables are established by measuring the physical characteristics of the chips. Hardware operation of the proposed system architecture is described as follows.

1. Desired clock frequency is determined by the proposed voltage scheduling method, which is to be explained in the next section.
2. Desired supply voltage is looked up from the device driver.

Top

3. Target processor sets these values into power controller by sending control codes. After that, target processor stops running, and waits while clock frequency and supply voltage are settling down to steady state. Duration of this transition time is looked up from the device driver.
4. Power controller changes clock frequency and supply voltage. After that, target processor restarts running

Low Power Fixed Priority Scheduling. Low Power Fixed Priority Scheduling (LPFPS) [11] is the power efficient version of the fixed priority scheduling which is widely used in hard real time design. This method obtains power reduction of a processor by exploiting the slack times inherent in the system and those arising from variations of execution times of task instances. This algorithm uses a runtime mechanism to use these slack times efficiently for power reduction for processors that supports a power down mode and can change the clock frequency and supply voltage dynamically.

LPFPS exploits both execution time variation and idle time intervals to obtain a power saving for a processor while ensuring that all tasks adhere to their timing constraints. To obtain the maximum power saving, LPFPS dynamically vary the speed of the processor whenever possible, and bring the processor to a power down mode when it is predicted to be idle for a sufficiently long interval. Specifically, if there is only one task eligible for execution and its required execution time is less than its allowable time frame, the clock frequency of the processor along with the supply voltage is lowered. If it is detected that there is no task eligible for execution until the next arrival of a task, the processor enters power-down mode. Both these mechanisms are made possible by a slight modification of the conventional fixed priority scheduler.

The fixed priority preemptive scheduler in the kernel can be implemented easily using runtime queues. Because most information about the tasks is available through queues and LPFPS depends on this information, the scheduler for LPFPS can be implemented with a slight modification of the conventional scheduler. The pseudo code for the LPFPS scheduler is shown below.

```
if current_frequency < maximum_frequency then
       increase the clock frequency and the supply voltage to the maximum value;
       exit;
end if
while delay_queue.head.release_time ≤ current_time do
       enqueue delay_queue.head in the run_queue;
end do
if run_queue.head.priority > active_task.priority then
       set the active_task.executed_time;
       perform context switch;
end if
if run_queue is empty then
       if active_task is null then
              set timer(delay_queue.head.release_time - wakeup_delay);
              enter power down mode;
       else  speed_ratio = Compute_speed_ratio();
              find a minimum allowable clock frequency
              /* this frequency must be greater or equal to the ratio: speed_ratio * max frequency*/
              adjust the clock frequency along with the supply voltage;
       end if
```

Top

Episode Based Algorithm. This algorithm [8] is implemented in the Linux kernel and requires no modification of user programs. Unlike previous automated approaches, this method works equally well with irregular and multiprogrammed workloads. In order to determine the right level of CPU performance (or CPU speed) this algorithm uses processes information (e.g. deadline) and process classification information (background, periodic, foreground) automatically from OS kernel. The algorithm focuses primarily on interactive applications.

This algorithm divides the application execution into episodes. There are two types of episodes: interactive and periodic, with producer and consumer subcategories, where the communications between these episodes determine their performance level. Episodes are triggered by communication events with specific tasks but multiple tasks may be involved during episode execution. 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. These classifications can be used to drive deadlines for the execution episodes and guides performance-setting decisions on a per-task and per-episode basis.

To find interactive episodes, algorithm keeps track of the set of tasks that communicate with each other as a result if a user initiated a GUI event e.g pressing a mouse button or a key. GUI controller is responsible of starting an interactive episode.

Producer and Consumer episodes form a special subcategory of periodic episodes, where the distance from producer to consumer establishes the performance level. The producer episodes can be stretched to the beginning of their associated consumer episodes.

This algorithm predicts performance differently for each type of episode. For interactive algorithms, the predictor computes the performance factor, which is the ratio of the desired execution speed and the processor's maximum speed. Incase of interactive episode, it is difficult to find the optimum performance since it depends on the user not on some event. So, the performance factor predictor for interactive episode works by starting with initial value, set to minimum performance factor of the processor, then refining its value. The algorithm uses the following three steps:

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

Based on the estimate of the episode execution time at full performance, the optimum performance level can be estimated for an interactive episode. For periodic episodes, the optimum performance factor can be computed by stretching the periodic episode's execution to the beginning of the next episode or to the beginning of the

Top

associated consumer. An important consideration is to find the performance factor when interactive episodes are present in addition to the periodic activity. The algorithm works as following:

1. When there is no interactive episode executing on the processor, the performance factor is set to the one computed for the periodic activity.
2. At the beginning of an interactive episode, the performance factor is switched to the one that was predicted for the task's interactive episodes, if it is higher than periodic performance factor.

Low-Power scheduling using Slack Estimation Heuristic. The system [12] emphasizes improving the on-line slack estimation part of a DVS algorithm and proves that a good slack estimation method can significantly improve the energy efficiency. The system proposes an on-line DVS algorithm for periodic real-time tasks that are scheduled under the *Earliest-Deadline-First* (EDF) algorithm. The system uses two notations to keep track of the available slack times of each task. U_i^{rem} denotes the unused execution time and W_i^{rem} denotes the remaining WCET of the task. The system assumes that a real-time scheduler has two queues: $waitQueue$ and $readyQueue$. The $waitQueue$ and the $readyQueue$ contain the completed tasks and the currently activated tasks, respectively. All the tasks are initially queued in $waitQueue$, in which the tasks are sorted by their next arrival time. When a task is activated, the task is moved from $waitQueue$ to $readyQueue$. At each task activation, both are set to w_i i.e., $U_i^{rem} = W_i^{rem} = w_i$. Among the tasks in $readyQueue$, the *active task* with the earliest deadline is scheduled to run under the EDF scheduling policy. As T_α executes, its W_α^{rem} decreases and consumes its available execution time. T_α may complete its execution or be preempted by a higher-priority task instance. When T_α is preempted by a newly activated higher-priority task instance, T_α is requeued into $readyQueue$ while waiting for the resumption. When T_α completes its execution, its remaining WCET W_α^{rem} is reset to 0, and T_α is inserted into $waitQueue$. It should be noted that the unused time is not reset. U_α^{rem} is used to estimate the slack time available for other task instances.

3. Comparison Criteria

To enable us discuss and assess existing systems of operating systems supporting power management in mobile devices, we propose the following general attributes for assessing, classifying, and comparing these systems. These are analyzing which type of state change strategy is supported by the system, the type of scheduling algorithm employed, the speed setting policy used, the CPU utilization prediction approach used and the type of tasks the system is optimized for. These criteria have been proposed based on the components of the DVS system. These criteria have been explained in the following lines.

3.1 State Change Implementation Class

The different levels of energy that can be consumed by a device are categorized as *states*. Applying DPM techniques results in the change of the state. A device can be made to enter any of the possible states. Thus, power conservation can be optimized by adjusting power parameters on-the-fly while ensuring realtime deadlines of running software are met [13]. In other words, it dynamically determines power states according to

workloads. Power usage is minimized by putting a device in idle state if it is not required by any task. The idle state has been referred to by other names including sleeping state and stand-by-state [14]. *Power managers (PM)* [14] determine transitions between power states according to the power management policies (also known as *algorithms* / *policies*) [15]. Power management can also be generalized for more than two states, besides the *sleep* and *awake* states. Each of these states has a different performance level and a corresponding power consumption level [14]. Changing the states requires time. T_{sd} and T_{wu} have been defined as shutdown and wakeup delays. A device should sleep only if the saved energy is more than the energy required to make the change. This is determined by the *policies* being used. Figure 2 gives an example of power management form the perspectives of the various factors involved in state change.

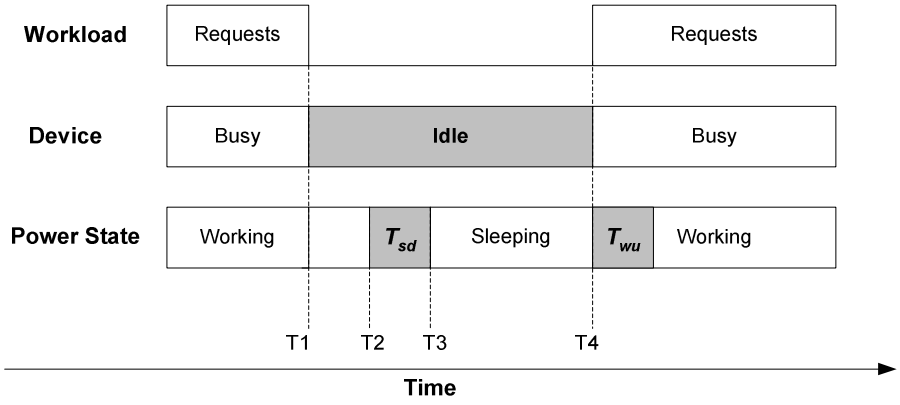

Figure 2. Power management and state change

The change of states can be done through implementation in hardware or device drivers i.e. at the system level. It can also be done entirely through operating system. A third approach is to implement them by employing a combination of the two techniques. These techniques have been described below.

System-Level Dynamic Power Management. The *power managers* observe the requests at devices and predict the future workload. Using the estimated workload a suitable *state* is decided for the device and then the device is switched to that particular state. The components can be managed either *internally* or *externally*. *Internally-managed* devices, also called *self-managed* components, use conservative policies. *Greedy* policy shuts down the processor as soon as an idle period is detected. These policies can be classified as predictive or stochastic. Predictive techniques can be further classified as static or adaptive techniques. This technique can be diagrammatically depicted as in Figure 3 [16]. It is evident that the power manager does not distinguish between requesters. Instead, it simply honors the requests it gets from the various *requesters / tasks*.

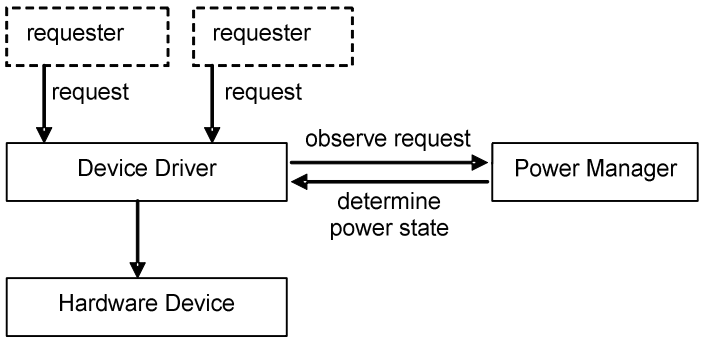

Figure 3. System-level power management

Operating System Directed Power Reduction. One of the techniques for managing the power of the system is through the OS [14]. The management depends on the *requesters* (currently running tasks), for changing the state of a device. Initial research has been conducted based on the observation of requests for a specific device. This mechanism is known as the *device-level power management* (DLPM). A better strategy proposed is to change the state according to the information provided by the requesters. This strategy has been referred to as the *task based power management* (TBPM). This strategy excels based on the fact that the processes provide more information about future requests to a specific device as different tasks can have different request patterns. Tasks can be created or terminated. Some tasks may have tighter performance requirements. TBPM considers the CPU time of tasks while deciding the power state changes. TBPM uses a two-dimensional data structure called the *device-requester utilization matrix,* **U**. It also uses a vector called the *processor utilization vector*, **P**. The elements of **U** are denoted by $U(d,r)$ where d is the utilization by a requester r. Also, **P** is the processor utilization for each process and $P(r)$ represents the percentage of processor time used by the requester r. Figure 4 shows the states in which a process can be at any time [17].

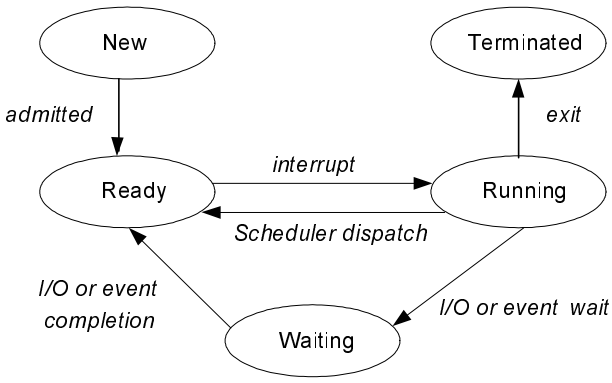

Figure 4. Process states

Whenever a new requester is created, a new column is added to the data structures and the respective column is deleted upon deletion or termination of the requester. The utilization is initially set to zero and is updated as and when required. The power manager periodically evaluates the utilization and makes appropriate changes.

Top

The utilization for any specific device is the sum of the utilization values for all the requesters of that device. Figure 5 shows the incorporation of TBPM into the process state diagram. This incorporation depicts allocating, deleting the column and updating the $U(d,r)$ and $P(r)$ data structures.

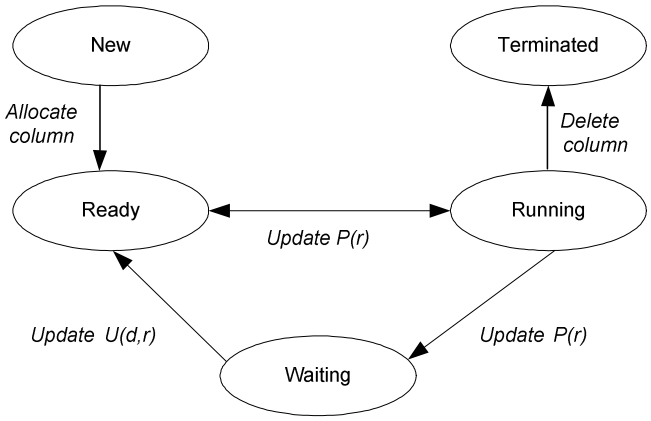

Figure 5. Incorporation of TBPM into process state diagram

Figure 6 shows the relationship between the *requesters/tasks*, the *device driver* and the hardware. It depicts the implementation in which the requester sets the power state of the device [16]. It is evident that the *tasks* or the *requesters* do not play any part in setting the state of the device.

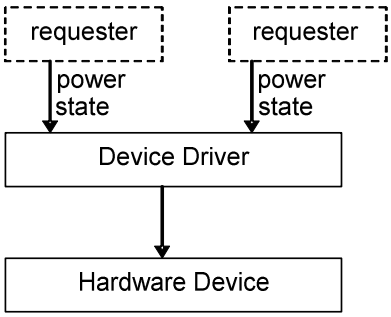

Figure 6. Tasks controlling the power state of the device

Requester-Aware Power Reduction. The *requesters / tasks* actively specify the devices they will require for execution. This is done through implementing an API [9,11]. The API is defined below.

$$RequireDevice(device, type, period, wait), \tag{8}$$

where, *device* denotes the hardware device, *type* specifies the nature of access of the device. This can be always or periodic or once or delete. The term *period* specifies the length of the time for which the device will be required in milliseconds and *wait* specifies the time that is allowed to wait.

Top

We can visualize the implementation of this technique in Figure 7 The power manager receives information from the scheduler and the process management module and based on this information determines the state of the device.

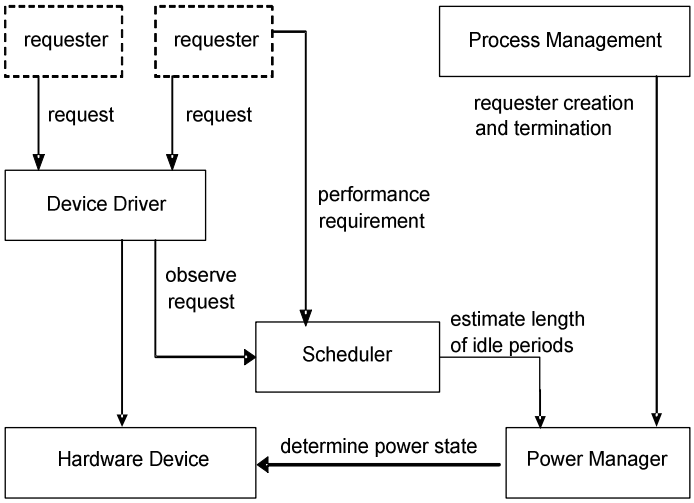

Figure 7. Power manager utilizing information from the scheduler and process manager

The design of the power management policies can determine the level of power consumption. Efficient policies optimize the power consumption of the system. The problem of finding an optimal policy to maximize the average performance level can be formulated as a stochastic problem [15]. And policy optimization can be solved in polynomial time. The most aggressive policy turns off every system component as soon as it becomes idle and is known as the *eager* policy.

Research has also been done to manage the power through the operating system [14]. Designing efficient policies for 8 have also been studied [15]. Templates have also been designed for controlling power management through software [18]. The template is implemented as a kernel-level filter driver (FD) that is attached to the device drivers from the vendors. The software that controls implements the policies/algorithms is known as the power manager (PM). These policies are sent to the FD to be implemented.

3.2 Scheduling Algorithms

Scheduling algorithms can be categorized on the basis of tasks they are supposed to schedule. We can classify the tasks as being either aperiodic or periodic. Aperiodic tasks are single tasks activated at irregular intervals whilst periodic tasks are a group of identical tasks activated at regular intervals.

Early Due Date (EDD). This algorithm selects tasks with earlier due date first. It has a complexity of O(nlogn), If after the application of the EDD algorithm, a feasible solution exists and unused slack is available, then the following algorithm is invoked to decrease the energy consumption. In each iteration, the voltage of the critical task (or task m) is decreased, the voltages of the other tasks adjusted, and the finishing time of the

task, *f*, compared with its deadline. If there is a violation of task *j* at step *k*, i.e. there is a violation in the assignment *Vj (k)*, then the previous voltage value (*Vj(k - 1)*) is the optimal voltage value, here V is the voltage supplied. Furthermore, since all tasks with earlier due date than task j could have caused deadline violation of task j, all these tasks are assigned voltages corresponding to iteration *k-1*, $V_l = V_l\ (k\text{-}1)$ for $l = 1$ *to* j. The algorithm continues until voltages for all the tasks are determined. The worst case complexity of the algorithm is $O(nk_{\max})$. This is because there are at most $k_{\max}$ iterations, and in each iteration, at most n task voltages are calculated. The pseudo code for this algorithm is given below.

```
for k = 1 to k_max
update V_m(k), T_m(k) for the critical task
for each unscheduled task
       update V_j(k), T_j (k) for the tasks
       f_j = f_j-1 + T_j(k)
       if f_j > d_j(k)      /* d_j is the deadline of task j*/
              for i = 1 to j
                     Schedule: V_i = V_i(k-1).
                     return /* end, if all the tasks are scheduled*/
       end if
```

Aperiodic Earliest Deadline First (EDF). This is a dynamic scheduling algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks. In this algorithm, whenever a new task arrives, the voltage values of the unscheduled tasks are updated provided that the deadline constraints are not violated. In each cycle, the task with the earliest deadline is scheduled with complexity $O(n\log(k_{\max}))$, using a binary search. The overall complexity of the algorithm is $O(n^2(\log(k_{\max}))$.

Rate Monotonic (RM). This is a scheduling algorithm assigns priorities to tasks according to their request rates. In addition, deadline of a task (d_j) is equal to its period. The sufficient but not necessary guarantee test is $U \le U_{\text{lnb}} = n(2^{1/n} - 1)$. In this algorithm, the voltage values of the unscheduled tasks are updated according to the minimum energy equation provided $U - U_{\text{ln}b}$. Here '*U*' is the utilization factor and it is calculated by the following equation.

$$U = \sum_{i=1}^{n} (Ti / Pi) \le 1, \tag{9}$$

where, *p* is the period of the task *i*, *n* is the number of tasks and (T_i / P_i) is the fraction of processor time spent in executing of task *i*. The worst case complexity of the algorithm is $O(n\log(k_{\max}))$. This is because at most $\log(k_{\max})$ iterations are needed, and in each of the iterations at most n calculations are done.

The Greediness Technique. According to [5], techniques are possible to identify and forcibly block processes that are not doing useful work for a set of period time. This is known as the greediness technique. Identification of process acting greedily is a major issue. Researchers have used three factors to decide if a process is not involved in active computing,

- If a device performs no I/O device read or write,

Top

- Does not have a sound chip on and
- Does not have the cursor appear as a watch.

When O.S. determines that a process is acting greedily, it blocks it for period called forced sleeping period, the issue here is a short sleep period saves insufficient power, while long sleep period may block a process that actually doing some thing useful.

3.3 Speed Setting Policies

Any DVS algorithm must consider speed setting policies that could strike balance between performance and energy efficiency. A speed setting policy must first predict the utilization of the processor in near future and then must take a decision regarding speed adjustment. In general, there are three prediction techniques 1) monitoring average CPU utilization at periodic intervals 2) using application worst case CPU demands 3) using application runtime CPU utilization. Below, we have summarized some of the speed setting policies that are widely discussed in the literature.

FLAT<speed>. This implies that voltage scaling would not be done. The operating voltage is fixed at a constant level scaling and further scheduling is disabled. It sets speed fast enough to complete the predicted new work plus the excess-cycles being pushed into the coming interval. This is subject to the limit of fullspeed = 1. This policy is weak on prediction because it simply tries to smooth speed to a global average. The parameter *speed* is the input variable and the range should be between 0 and 1. The speed is normally set fast enough to complete at least the excess-cycles so that no work takes more than one interval.

COPT. This is the theoretical optimum operating point generated by post-simulation trace analysis and is used for comparison with realizable algorithms. COPT (*clipped optimal*) is the minimum energy at which a system can obtain a given delay.

PAST. PAST calculates how busy the last completed interval was (including excess-cycles brought into that interval). It then predicts that the coming interval will be equally busy. If the prediction is for a busy interval, PAST increases speed; if for a mostly idle interval, PAST decreases speed. Some smoothing is accomplished by limiting the amount by which speed can change (except that speed may be increased to 1 if excess- cycles rises particularly high).

AVG<*weight*>. This algorithm computes an exponentially moving average of the previous intervals. At each interval the run-percent from the previous interval is combined with the previous running average, forming a long-term prediction of system behavior. The term *weight* is the relative weighting of past intervals relative to the current interval (larger values mean a greater weight on the past) using the equation. Empirical comparison of the performance of these scheduling algorithms has also been done in [19].

$$weight = (weight \times old + newweight + 1), \tag{10}$$

Top

3.4 Prediction Approach

As discussed earlier, DVS systems can be classified to be comprised of three main parts. One of these is the component that determines the demand of the processor by various tasks. Its main job is to predict the usage of the processor by the task. This prediction must be very accurate otherwise the system might suffer from *over-prediction* or *under-prediction*. For this purpose, the prediction approach can be categorized into two main types. These are *static* and *adaptive*. The *static* techniques can be further classified into *fixed timeout* and *predictive shutdown*. The timeout strategy has two main advantages. They are very general and the level can be altered by changing the values of the timeout factor. This feature can also work against the strategy in the sense that large timeouts will cause a higher degree of *under-prediction* or *over-prediction*. Two predictive schemes have been proposed in [20]. The first approach develops a nonlinear regression equation based on the past history. The second approach is based on a threshold. The adaptive techniques can be distinguished between using simple average of the CPU utilization at regular intervals or using the application's worst-case CPU demands or using the application runtime CPU usage. The first two approaches are heavily dependent on dynamic input.

3.5 Type Of Tasks Optimized

Tasks can be classified as periodic and aperiodic. Periodic tasks are a group of tasks that occur at regular intervals while aperiodic tasks do not follow any fixed interval. Scheduling for these two types is slightly different as some algorithms look at the interval of time in which the tasks arrive. Based on this interval the algorithm calculates the priority of the tasks. Aperiodic tasks can be selected using an arbitrary queuing discipline. The periodic tasks are normally of higher priority as compared to the aperiodic tasks. Also, the priority of aperiodic tasks constantly changes as these tasks get scheduled. This is because aperiodic tasks utilize the processor for different lengths of time in different time quantum. Aperiodic tasks are also known as sporadic tasks.

4. Comparison

The qualitative comparison of the DVS systems described in Section 2 has been done based on the parameters discussed in Section 3. Each system was evaluated based on the criteria and the results have been summarized in Table 1. Since in this paper we have discussed operating system role in energy state adjustment, we have only considered systems that employs operating system directed power management. We have also judged the DVS systems according to the scheduling algorithms they use. Two of the systems use EDF, one uses offline scheduling, whereas two use their modified versions of EDF. Speed scaling policy is also considered in the comparison table and algorithms are considered according to the speed scaling policy they employ. Three of the systems are designed for hard real time systems while the remaining three are for soft real time systems. All the systems except LPFPS use adaptive approaches for predicting the future clock cycle requirements of the task. LPFPS on the other hand uses static prediction technique. The systems are also analyzed according to the type of tasks that they expect. We have categorized these tasks as periodic and aperiodic.

Top

Table 1. Comparison of DVS Systems

Metrics / Systems	State Change Implementation	Scheduling Algorithm	Speed Scaling Policy	Prediction Approach	Types of Tasks Optimized
Grace OS	Operating system directed	EDF. Soft real time	Implements function $schedule(x, y)$ where y is speed and x is cycles	Using application runtime CPU usage, Stochastic	Periodic
RVH	Device level technique incorporated with operating system by dividing the tasks into timeslots	Utilizes workload-variation slack time by partitioning a task into several pieces, then dynamically controlling supply voltage on timeslot-by-timeslot basis for SRT	Voltage scaling is done by using device driver lookup tables established by measuring the physical characteristics of the chips.	Utilizes the workload slack time of the tasks, Predictive	Both
LPFPS	Operating system directed	It schedules tasks by exploiting the slack times inherent in the system and those arising from variations of execution times of task instances for HRT	Heuristic approach is employed using best and worst case execution time.	Exact value is known beforehand, and therefore there is no need to predict any value. Offline table	Periodic
Episode based Algorithm	Operating system technique that uses requester directed technique	Schedules tasks by monitoring their online usage for SRT	Speed is scaled differently for interactive and periodic tasks. Interactive tasks are given preference. Speed is scaled as a function of peak supply voltage.	Predictor computes the performance factor, is the ratio of the desired execution speed and the processor's maximum speed.	Both
Lp/SEH	Operating system directed	EDF, HRT, preemptive	Speed is set as a function of maximum supply voltage.	Own algorithm for estimating online slack time due to dynamic workload variation	Periodic

5. Conclusion

This study compares the DVS systems by considering the components that constitute a complete DVS system. The most important components are the components that implement the scheduling algorithm and the component that implements the speed scaling policies. We have analyzed the presented systems according to these components. Grace OS system implements function *schedule (x,y)* to change the speed and uses the application runtime CPU usage. It uses stochastic model to predict the idle time. Tasks in Grace OS are periodic (occur after specific time periods.) Also it supports state change to make the processor run at low power. RVH controls the speed of processor by using device driver lookup tables which are established by measuring the physical characteristics of the chips. It supports periodic and not periodic tasks and predicts the idle time by utilizing the workload slack time of the tasks. The mechanism of scheduling algorithm used in RVH is partitioning a task into several pieces and then dynamically changing the voltage on timeslot basis for SRT. LPFPS system controls the task scheduling by exploiting the slack times inherent in the system and those arising from variations of execution times of task instances for HRT. This system supports state change and use periodic tasks. To predict the idle time it uses the exact value known beforehand. Episode-based algorithm schedules the tasks by monitoring their online usage for SRT and use a factor called performance factor (PF) to predict idle time that is the ratio of the desired execution speed and the processor's maximum speed. It can work for periodic and not periodic tasks. Lp/SEH system uses EDF, Hard Real Time and preemptive techniques for scheduling and also supports periodic tasks. The system has its own algorithm for estimating online slack time due to dynamic workload variation. The preference of one system over the other depends on factors specific to the type of tasks the device is scheduling and also the usage of the device. It also depends on some hardware parameters. Based on these observations, no system can be classified to be the best.

Our comparison provides a clear view of the components used to configure a complete DVS system. This analysis can be employed to tailor new DVS systems that correspond to specific need. It has been studied that most of the systems use EDF algorithm or a variant of it to schedule tasks. It has been noted in literature that RM algorithm also fares well and hence can be used instead of the EDF algorithm.

The technique of the tasks requesting specific devices by sending certain parameters can also be extended. For this purpose, the scheduling algorithms and the speed scaling policies should be optimized to determine states corresponding to the processor provided states. The parameters in $RequireDevice(device, type, period, wait)$ API can be used to compute the time for which a specific task would execute and based on that the corresponding state of the processor can be chosen. The scheduler would then schedule together all tasks requiring the same state.

The concept of a software template being used to control power state changes can be extended to enable the user to select the scheduling algorithm, the speed setting policy and also the prediction approach. A matrix should be implemented that proposes the optimal system. A *weight* should be associated with each solution. This *weight* should be periodically re-calculated based on stochastic Markov techniques. The proposed speed should have a direct mapping with the states supported by the processor. This will be very useful in case there

Top

are tasks running which have either fixed or dynamic priority, i.e. the set of tasks is a combination of realtime tasks and normal tasks.

References

[1] G.H. Forman and J. Zahorjan, “The Challenges of Mobile Computing,” *IEEE Journal on Computer,* vol. 27, no. 4, April 1994, pp. 38-47.

[2] M. Satyanarayanan, “Fundamental Challenges in Mobile Computing,” *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, Pennsylvania, United States, 1996, pp. 1-7.

[3] T. Imielinski and B. R. Badrinath, ”Mobile wireless computing: Challenges in data management,” *Communications of the ACM*, vol. 37, no. 10, October 1994, pp. 18-28.

[4] M. Weiser, B. Welch, A. Demers, S. Shenker, Xerox PARC, “Scheduling for Reduced CPU Energy,” *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.

[5] J. R. Lorch and A. J. Smith, “Scheduling techniques for reducing processor energy use in MacOS,” *Wireless Networks*, vol. 3, no. 5, 1997, pp. 311-324.

[6] Y. H.Lu and G. D. Micheli, “Comparing System-Level Power Management Policies,” *IEEE Design & Test,* vol. 18, no. 2, 2001, pp. 10-19

[7] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Norwell, MA: Kluwer, 1998.

[8] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic Performance Setting for Dynamic Voltage Scaling," *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking* (MobiCom'01), May 2001.

[9] W. Yuan, K. Nahrstedt, “Energy Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems” *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003

[10] S. Lee and T. Sakurai, “Run-time voltage hopping for low-power real-time systems,” *Proceedings of the 37th conference on Design automation*, Los Angeles, California, United States, 2000, pp. 806-809.

[11] Y. Shin and K. Choi, “Power conscious fixed priority scheduling for hard real-time systems,” *Proceedings of the 36th ACM/IEEE conference on Design automation*, 1999, pp. 134-139.

Top

[12] W. Kim, J. Kim, S. L. Min, "A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis", *Proceedings of the conference on Design, automation and test in Europe*, March 2002.

[13] Dynamic Power Management, *"Source Forge,"* http://dynamicpower.sourceforge.net/

[14] Y. H.Lu, L. Benini and G. D. Micheli, "Operating System Directed Power Reduction," *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, Rapallo, Italy, 2000, pp. 37-42.

[15] G. A. Paleologo, L. Benini, A. Bogliolo and G. De Micheli, "Policy Optimization for Dynamic Power Management," *Proceedings of the 35th annual conference on Design automation conference,* San Francisco, California, United States, 1998, pp. 182-187.

[16] Y.H. Lu, L. Benini and G.D. Micheli, "Requester-Aware Power Reduction," *Proceedings of the 13th international symposium on System synthesis*, Madrid, Spain, 2000, pp. 18-23.

[17] A. Silberschatz ,P. B. Galvin and G. Gagne, *Operating System Concepts*, Addison-Wesley, International edition, 2003.

[18] T. Pering, T. Burd, and R. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, San Diego Bay, San Diego, 1998.

[19] Y. H. Lu, T. Simunic, G. De Micheli, "Software Controlled Power Management," *Proceedings of the seventh international workshop on Hardware/software codesign*, Rome, Italy, 1999, pp. 157-161.

[20] M.B. Srivastava, A.P. Chandrakasan and R.W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, March 1996, pp. 42-55.

Top