Dynamic Voltage Frequecy Scaling for Multi-tasking Systems Using Online Learning *

Gaurav Dhiman
Dept. of CSE
University of California, San Diego
gdhiman@cs.ucsd.edu

Tajana Simunic Rosing
Dept. of CSE
University of California, San Diego
tajana@ucsd.edu

ABSTRACT

This paper presents an extremely lightweight dynamic voltage and frequency scaling technique targeted towards modern multi-tasking systems. The technique utilizes processors runtime statistics and an online learning algorithm to estimate the best suited voltage and frequency setting at any given point in time. We implemented the proposed technique in Linux 2.6.9 running on an Intel PXA27x platform and performed experiments in both single and multi-task environments. Our measurements show that we can achieve the maximum energy savings of 49% and reduce the implementation overhead by a factor of 2 when compared to state of the art techniques.

Categories and Subject Descriptors

J.6 [**Computer Applications**]: Computer-Aided Engineering

General Terms

Algorithms,Measurements,Experimentation

Keywords

Dynamic Voltage Frequency Scaling, Online Learning

1. INTRODUCTION

Power consumption is a key issue in the design of computing systems today. While battery driven systems need to meet an ever increasing demand for performance with a longer battery life, high performance embedded systems contend with issues of heating. Dynamic voltage and frequency scaling (DVFS) is a highly effective technique for reducing system power dissipation. The key idea behind DVFS techniques is to dynamically scale the supply voltage level of the CPU so as to provide "just-enough" circuit speed to process the system workload, thereby, reducing the energy consumption (which is quadratically dependent on the supply voltage level). A number of modern processors such as Intel's XScale [1], AMD Athlon [2] and Transmeta's Crusoe [3] are equipped with the DVFS functionality.

A DVFS technique that ensures that all the tasks meet their deadline requires critical information about all tasks, such as the task arrival time, deadline, and workload, to be known in advance. Such information is difficult to obtain in a general purpose multi-tasking systems such as Linux, Windows CE etc that are commonly used in modern embedded systems. In this paper, we propose a DVFS technique for such a general purpose multi-tasking environment. The basic premise is to design a control algorithm that selects among a set of voltage-frequency (v-f) settings to select the best suited setting at any given point in time in accordance with the task characteristics. The control algorithm bears the responsibility for accurately characterizing the current task's behavior and accordingly selecting the appropriate v-f setting. We employ an online learning algorithm [4] to perform this control activity. The online learning algorithm (referred to as "controller") has a set of v-f settings (referred to as "expert") to choose from and selects an expert which is most likely to minimize both the energy consumption and performance delay based on the characteristics of the currently scheduled task and the user preference in terms of energy-performance delay (e/p) tradeoff. The controller utilizes the run time statistics made available by the platform such as cache hit/miss ratio etc to perform this characterization. The design takes the multi-tasking environment into account and stores the task characterization information on a per task basis, which ensures that the updates and expert selection are made using the current task's information only. Such a design allows the technique to work accurately and seamlessly across context switches. The advantage of using an online learning algorithm for this purpose is that it provides a theoretical guarantee on the overall performance converging to that of the best performing expert.

We implemented the proposed technique in Linux 2.6.9 running on an Intel PXA27x platform. We experimented with benchmarks with varying characteristics and tested for both single task as well as multi-task scenarios. We calculated actual energy savings by performing current measurements in hardware. Using the technique, we achieved energy savings of up to 49%. The technique is extremely lightweight, and in our experiments we observed the overhead caused by it to be around a factor of 2 less than state of the art techniques.

*This research was funded by HPWREN under NSF grant numbers 0087344 and 0426879 (http://hpwren.ucsd.edu), Center for Networked Systems (http://cns.ucsd.edu)and Sun Microsystems

ISLPED'07, August 27–29, 2007, Portland, Oregon, USA.

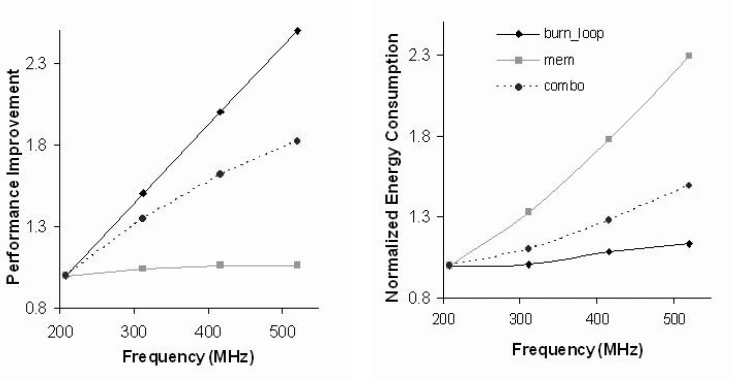
2. PREVIOUS WORK

DVFS has been an active area of research and a number of techniques have been proposed in the past. Previous DVFS-related work may be broadly divided into three categories. The first category of techniques target systems, where the task arrival times, workload and deadlines are known in advance [4, 5, 6, 7]. It is assumed that the total number of CPU cycles needed to complete each task is fixed and known a priori. DVFS is performed at task level by the OS in order to reduce energy consumption while meeting hard timing constraints for each task.

The second category of techniques require either application or compiler support for performing DVFS. In [8], the application code is divided into blocks and the worst-case execution time of each block is used to select voltage for the next block. In [9], the application is divided into slots and the voltage setting for upcoming slot is calculated with help of a software loop. In [10], a checkpoint-based algorithm is proposed in which the scaling points are identified off-line by the compiler. In [11], a compiler-assisted DVFS technique is proposed, in which frequency is lowered in memory-bound regions of a program. In [12], a DVFS technique for multimedia applications is proposed in which scaling is performed at each video frame based on the timing information given by the content provider. In [13], an intra-task scheduling method for a multiprocessor system is proposed where tasks are scheduled based on a predefined schedule set during the compilation step.

The third category comprises of system level DVFS techniques that make no assumptions about task characteristics or any support from compiler. These approaches rely on runtime statistics made available by the platform or micro-architecture to model the task behavior. In [14] a micro architecture-driven DVFS technique is proposed in which cache miss drives the voltage scaling. In [15] IPC (instruction per cycle) rate of a program execution is used to direct the voltage scaling. However, the results in these two works are based on simulations. In [16, 17, 18], dynamic runtime statistics such as cache hit/miss ratio and memory access counts obtained from a performance monitoring unit (PMU) (on an XScale platform) are used to determine the appropriate voltage setting. The policy in [16] uses pre-defined optimal frequency domains in a 2-D MPC (memory requests per cycle) and IPC space to perform scaling. Although this technique does take multi-tasking into account, it is not flexible in the sense that frequency domains are obtained empirically through experiments on micro-benchmarks for a given performance loss. It does not allow any run time control of e/p trade-off. In [17] and [18] a regression-based method is used to identify the degree of memory intensiveness of a task. The primary difference between them is that they use different platforms for experimentation and employ different variables for constructing the regression equation. Scaling is performed on the basis of this information and the user specified e/p tradeoff preference. Thus, these approaches dynamically model the workload and enable more precise control over energy performance tradeoff using the user preference. However, they present results only in a single task environment. Meanwhile, the approach in [19] uses an adaptation of the same online algorithm as we use, but for dynamic power management (DPM).

The primary contributions of our work are as follows: (1) We present a complete system level implementation and results of a DVFS policy in a multi-tasking environment. (2) Our approach provides a mechanism to enable control over e/p tradeoff. (3) It is based on an online algorithm which guarantees convergence to the best suited v-f setting. (4) The policy is extremely lightweight and has negligible overhead.



(a) Performance (b) Energy

Figure 1: Performance Improvement and Normalized Energy Consumption

3. TASK CHARACTERIZATION

DVFS problem is one of accurately characterizing the executing task, since it directly determines potential benefits of performing DVFS. Weissel et al [16] show that, if an executing task is CPU and cache intensive, the performance improvement scales linearly with increasing frequency. However, if a task is memory intensive, the performance improvement is relatively insensitive to increase in frequency. To verify this, we performed simple experiments with 3 tasks with differing characteristics: (1) Task *burn_loop*, which is highly CPU intensive, continuously loops to burn CPU cycles without accessing any memory location; (2) Task *mem*, which is highly memory bound, copies data from one memory location to another; (3) Task *combo* is a mix of the first 2. We performed our experiments on an Intel PXA27x platform, and operated the 3 tasks at 4 different v-f settings, namely (208MHz, 1.2V), (312MHz, 1.3V), (416MHz, 1.4V) and (520MHz, 1.5V). Figure 1a shows the performance improvement with increasing clock speed. As expected, *burn_loop* has a linear performance improvement with an increase in frequency while *mem* shows only a marginal improvement since its execution speed is limited by memory accesses. For *combo* task the improvement is about an average of that of *burn_loop* and *mem* since it has both memory intensive and CPU intensive phases.

Figure 1b shows the energy consumption of the tasks normalized against the lowest energy consumption. The energy savings of *burn_loop* task does not increase significantly with decreasing clock speed. The reason for this is the high performance delay of *burn_loop* at lower frequencies (Figure 1a) and the constant overhead of periodic kernel activities in addition to the CPU-intensive task itself, which increases with the decreasing clock speed and hence offsets the savings due to low voltage. However, for the *mem* task the gain in energy savings is significant with decreasing frequency since

its performance delay is low even at low frequencies (Figure 1a). The energy savings for *combo* are a mix of the other two. Clearly, CPU-intensive tasks do not gain much from running at low frequencies. Conversely, it is beneficial to run a memory bound tasks at a lower frequency. Typically tasks comprise of CPU or memory-intensive phases, much like our *combo* task. It is very difficult to identify these phases offline, since one cannot accurately predict cache, TLB misses or branch mispredictions. Hence, the problem of performing DVFS becomes one of identifying CPU-intensive and memory-intensive phases dynamically at run time.

In order to accomplish that task we use Cycles Per Instruction (CPI) stack measure. CPI stacks break down processor execution time into a baseline CPI plus a number of miss event CPI components [20]. The base CPI represents the inherent execution component of the workload, while the miss event CPI components reflect the lost cycle opportunities because of miss events such as cache and TLB misses, branch mispredictions etc. The following equation represents the average CPI in terms of its' CPI stack components:

$$\text{CPI}_{avg} = \text{CPI}_{base} + \text{CPI}_{cache} + \text{CPI}_{tlb} + \text{CPI}_{branch} + \text{CPI}_{stall} \quad (1)$$

In order to dynamically characterize each executing task, we construct its CPI stack at runtime by using the performance monitoring (PMU) of the Intel PXA27x developer's kit [1]. The PMU is an independent hardware unit with four 32-bit performance counters that can be used to monitor any four out of 20 unique events available simultaneously. We monitor the number of instructions executed (INST), data cache misses (DCACHE), cycles instruction cache could not deliver instruction (ICACHE) and cycles processor is stalled due to data dependency (STALL). We also get the total number of clock cycles (CCNT) elapsed since the PMU was started in order to calculate the CPI components:

$$\text{CPI}_{avg} = \text{CCNT}/\text{INST},\ \text{CPI}_{dcache} = (\text{DCACHE} \times \text{PEN})/\text{INST}$$
$$\text{CPI}_{stall} = \text{STALL}/\text{INST},\ \text{CPI}_{icache} = \text{ICACHE}/\text{INST} \quad (2)$$

where CPI_{cache} has been broken down into CPI_{icache} and CPI_{dcache} and PEN is the average miss penalty for a data cache miss. Note that CPI_{tlb} and CPI_{branch} are missing. This is because we can monitor only 4 events at a time, and in our experiments we found the events being monitored more indicative of the task characteristics. Hence, we can estimate CPI_{base} as follows:

$$\text{CPI}_{base} = \text{CPI}_{avg} - \text{CPI}_{icache} - \text{CPI}_{dcache} - \text{CPI}_{stall} \quad (3)$$

We next define μ as a ratio of CPI_{base} to CPI_{avg} (equation 4) in order to measure the CPU-intensiveness of each executing task. For example, the task *burn_loop* has μ close to 1 since it executes on the CPU most of the time, while the task *mem* has much lower value of μ due to numerous memory related stalls it causes.

$$\mu = \text{CPI}_{base}/\text{CPI}_{avg} \quad (4)$$

During normal execution tasks have both CPU-intensive and non CPU-intensive phases. As a result, we need to dynamically estimate μ. A natural place to implement that is with the OS scheduler ticks (10ms for Linux 2.6.9 we used). For every scheduler quantum we monitor events using PMU, construct the CPI stack and perform μ estimation using equations 2, 3 and 4 (we use PEN=50 cycles in equation 2). We then use μ as an input into our online learning algorithm that then determines the appropriate voltage-frequency setting for the upcoming scheduler quantum. The next section outlines our implementation of the algorithm.

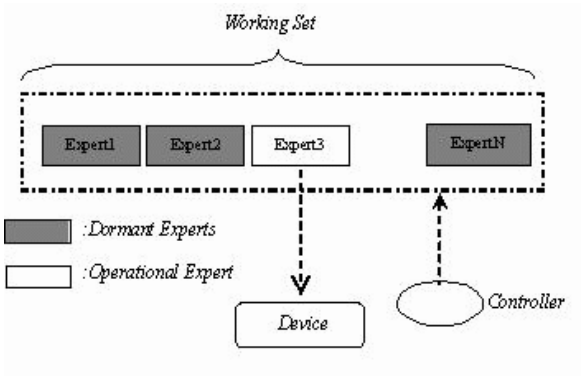


Figure 2: System Model

Table 1: Algorithm Controller

Parameters: $\beta \in [0, 1]$
Initialization:
-Weight vector $\mathbf{w}^1 \in [0, 1]^N$ of new task
-Initialize PMU
-Evaluate μ-mapper and μ-means for all experts
For scheduler ticks $t = 1, 2 \ldots$

1: Calculate μ
2: Update the weight vector of current task: $w_i^{t+1} = w_i^t \cdot (1 - (1 - \beta) \cdot l_i^t)$
3: Choose expert with highest probability factor in r^t, where $r^t = \frac{w^t}{\sum_{i=1}^{N} w_i^t}$
4: Apply v-f setting corresponding to operational expert to CPU
5: Reset and restart PMU

4. ONLINE LEARNING FOR DVFS

The online learning framework, as shown in Figure 2, consists of three entities: controller (the core online learning algorithm), experts (the v-f settings available) and the CPU. The set of experts is collectively referred to as the "working set". An expert is any allowable v-f setting supported by the platform. Each scheduler tick only one "operational expert" is selected by the controller to determine the v-f setting while the rest become "dormant experts". Our controller uses an online learning algorithm which is guaranteed to converging to the best performing expert for any workload. The convergence rate is a function of T, the number of scheduler ticks, and N, the number of experts ($O(\sqrt{\ln N/T})$[21]).

Table 1 has controller's pseudo-code. There are N experts (v-f settings) to choose from; $i = 1, 2 \ldots N$. The algorithm associates and maintains a weight vector $\mathbf{w}^t$ for the experts, $\mathbf{w}^t =< w_1^t, w_2^t \ldots w_N^t >$. Individual per task weight factors, w_i^t, reflect the suitability of their corresponding expert for the task. At initialization we assign equal weights to

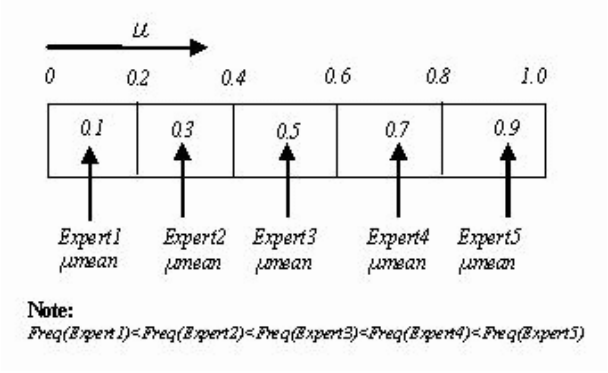


Figure 3: Sample μ-mapper

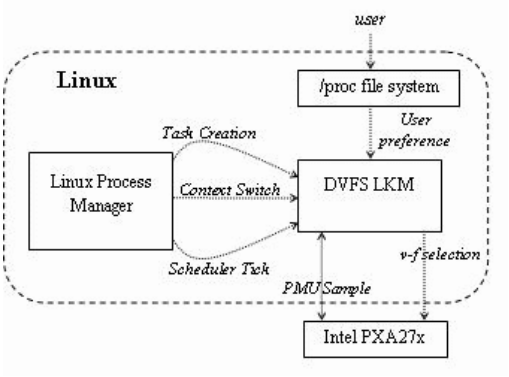


Figure 4: DVFS System Level Implementation

Table 2: Loss Evaluation Methodology

Value of μ	Energy Loss (l_{ie}^t)	Perf Loss (l_{ip}^t)
$\mu > \mu$-mean	0	($\mu - \mu$-mean)
$\mu < \mu$-mean	(μ-mean $-\mu$)	0
Total Loss (l_i^t) $= \alpha \cdot l_{ie}^t + (1-\alpha) \cdot l_{ip}^t$		

all the experts, start up PMU and set up μ-mapper and μ-means. Figure 3 shows a sample μ-mapper for 5 experts, where the frequencies increase in equal steps from 0 to Expert5. The mean of each interval, μ-mean, is associated with each respective expert, and is used for performing the per task weight update, w_i^t. In section 3 we have seen that CPU-intensive tasks should run at a high frequency, a non CPU intensive tasks at a low frequency, and tasks which are a mix of the two at an intermediate frequency. Therefore we estimate that the appropriate frequency for any fragment of a task scales linearly with its μ.

On each scheduler tick the controller reads the values being monitored from the PMU and calculates μ using equations 2, 3 and 4 (step 1 in Table 1). On the basis of this current value of μ, controller updates the weight vector of the currently scheduled task (step 2 in Table 1).

In order to perform this update, the algorithm first needs to evaluate how close an expert is to the best frequency for the current μ. The suitability (or rather unsuitability) of an expert for the current task is represented by a loss factor ($0 \leq l_i^t \leq 1$) which can be easily evaluated by comparing μ of the task to the μ-mean of each expert. We define both the energy loss (l_{ie}^t) and the performance loss (l_{ip}^t) as shown below, and use them to calculate the overall loss (l_i^t). If current μ is smaller than the expert's μ-mean, then the task is more memory intensive with respect to the given expert and hence can afford to run slower. It therefore has no performance loss, but since it could have saved energy by running slower, it has energy loss. Similarly, there is a performance loss, but no energy loss when $\mu > \mu$-mean. Table 2 summarizes how we evaluate the loss factor.

The 'α' factor in Table 2 is a user defined value that determines the relative importance of performance delay vs. energy savings. Once the loss factors are evaluated for each expert, the controller updates the weights of all the experts using the equation 5 (step 2 of Table 1). The value of β can be set between 0 and 1. The criterion for selecting the appropriate value is explained in [21]. For our experiments we used $\beta = 0.75$.

$$w_i^{t+1} = w_i^t \cdot (1 - (1-\beta) \cdot l_i^t) \tag{5}$$

At any point in time, the current weight vector accurately characterizes the task it belongs to, since it encapsulates all the updates based on previous μ values. This property of the weight vector obviates the overhead of storing previous PMU samples for online estimation of task characteristics, as done in some previous work [18]. To perform expert selection, the controller maintains a probability vector $\mathbf{r}^t =< r_1^t, r_2^t \ldots r_N^t >$ where $0 \leq r_i^t \leq 1$ are probability factors associated with each expert for the scheduler tick number 't'. The $\mathbf{r}^t$ is obtained by normalizing the weight vector as shown below:

$$\mathbf{r}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t} \tag{6}$$

At any point in time, the best performing expert has the highest probability factor amongst all the experts and hence the controller simply selects the expert with the highest probability factor as the operational expert for the next scheduler quantum (step 3 in Table 1). Once the operational expert has been chosen, the corresponding v-f setting is applied to the CPU (step 4 in Table 1). Lastly the controller restarts the PMU so that μ for the upcoming scheduler quantum can be evaluated at its conclusion (step 5 in Table 1).

5. EXPERIMENTAL RESULTS

We implemented our algorithm as a Linux 2.6.9 loadable kernel module (LKM) on Intel PXA27x platform. As shown in Figure 4, the LKM is closely knit to the Linux process manager. The Linux task data structure *task_struct* is modified to include the weight vector to support accurate per task characterization in a multi-tasking environment. The process manager notifies the LKM of task creation (which is used to initialize the weight vector) and scheduler tick occurrence (at which point it runs the algorithm shown in Figure 1) and context switch. The LKM also exposes a */proc* interface to the user, which is used to specify the e/p tradeoff (α).

CPU energy savings are calculated by current measurements using a 1.25M samples/sec DAQ. In our experiments we use a working set comprising of experts listed in Table 3(a). The frequency of each expert increases in steps of

Table 3: DVFS Working Sets

Freq (MHz)	Voltage (V)
208	1.2
312	1.3
416	1.4
520	1.5

(a)

Freq (MHz)	Voltage [18] (V)	Voltage [16] (V)
333	0.91	1.0
400	0.99	1.1
466	1.05	1.1
533	1.12	1.1
600	1.19	1.3
666	1.26	1.3
733	1.49	1.5

(b)

104MHz and corresponds to the highest frequency available at the associated voltage. Including experts with intermediate frequencies does not offer any advantage in terms of energy efficiency, since we cannot regulate the voltage at a finer granularity. Table 3(b) shows the experts available on another platform similar to ours in 2 variants: one with a finer granularity of voltage levels [18] and other with coarser granularity [16]. Fine granularity of voltage levels provides a great deal of advantage to the DVFS technique. For instance, in [16], energy savings for the well know UNIX utility gzip is reported to be 40% at 333MHz and 35% at 466MHz when compared against the baseline energy consumption at 733MHz. However in [18], for the same benchmark, the energy savings is in excess of 70% at 333MHz and around 65% at 466MHz. This observation highlights the advantages of a finer granularity voltage levels in the platform. In our platform, the voltage levels are coarse, and as we will see in the results that even with static scaling we cannot achieve higher energy savings than 54%.

We experimented with a number of applications with and without the DVFS LKM, in both single and multitasking environments. The energy savings and performance delay values are compared to a system running at 520MHz/1.5V. The chosen applications include common UNIX utility gzip for decompression (dgzip) and 3 representative benchmarks taken from an open source benchmark suite mibench [22]: bf (blowfish) - a symmetric block cipher with a variable length key from 32 to 448 bits; djpeg - decoding a jpeg image file and qsort - sorting a large array of strings in ascending order. The results achieved under both single and multi task environments for these benchmarks are illustrated in Table 4. We discuss them below separately.

Single Task Environment: Table 4(a) displays the results we achieved for each individual benchmark in a single task environment. The %energy indicates the amount of energy saved and %delay shows the amount of performance delay caused relative to the case where we do not have the DVFS LKM. As explained before, the α factor represents the user specified e/p tradeoff preference. In our experiments we tested with values of α ranging from 0.3 (low) to 0.7 (high) and used value of α of 0.5 for the medium value. From the results we can observe that as we increase the value of α, we get higher energy savings and for lower values of α, we get low performance delays. For instance, with qsort, the delay is just 6% for a low value of α, while the energy savings are 41% for a high value. From our offline analysis of these benchmarks, we estimate maximum possible energy savings on our platform with the given working set for qsort, djpeg, dgzip and bf to be 48%, 54%, 54% and 51% with performance delay of 56%, 34%, 33% and 40% respectively (at 1.2V/208MHz). Hence, the energy savings for all the benchmarks is on an average within 8% of the maximum possible at much lesser overhead for high α.

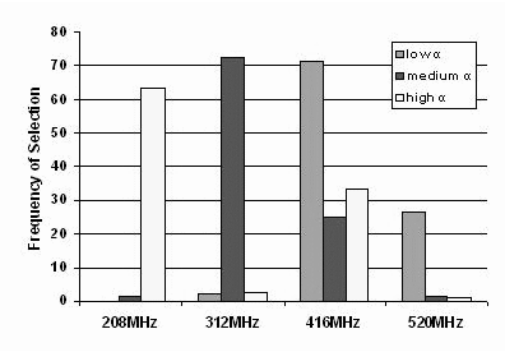


Figure 5: Frequency of selection of experts for qsort

Figure 5 shows the frequency of selection of different experts for qsort according to the selected value of α. We can observe that for higher value of α, the 208MHz expert is selected for around 65% of the time, while for the rest of the time, the 416MHz expert is chosen. This shows that the controller is able to identify both the memory bound phases (which are in majority for qsort) as well as CPU bound phases and accordingly select the best suited expert. Hence, we can see that α factor offers us a simple yet powerful control knob to obtain the desired e/p tradeoff.

Multi Task Environment: We experimented with the benchmarks in a multi-tasking environment as well to verify, if our per task characterization worked accurately. We performed these test by spawning 2 threads running different benchmarks simultaneously. Table 4(b) presents the results we achieved for multitasking for different values of α. From the results we can observe that as we increase the value of α, we get higher energy savings and for lower values of α, we get low performance delays across all the benchmark combinations. For djpeg+dgzip, we observe, that the results are an average of the individual results in Table 4(a). This is to be expected since we store the weight vectors on per task basis to preserve task characteristics across context switches. The combined two task result is an average of each individual result since the duration of execution of both the individual tasks is equal. For qsort+djpeg, we observe that the results for all the values of α correspond very closely to the results of qsort in Table 4(a). This is because of the fact that the total time of execution of qsort benchmark is roughly 4 times the duration of djpeg benchmark. Hence, the total energy savings and delay values converge to that of individual qsort benchmark results. However, we observe in our experiments that the threads executing the 2 benchmarks have the same priority and that the djpeg benchmark runs exactly twice longer with qsort than alone. This implies that accurate preservation of characteristics enables djpeg to select the same experts as it does when running alone, hence keeping its effective run-time the same, irrespective of the context switches. We observed similar behavior for qsort+dgzip as well.

Overhead: The LKM adds overhead to the system, since it processes the 3 events delivered to it by the linux process manager as discussed in Figure 4. For measuring the overhead caused by the LKM, we use lmbench [23]. Specifically, we use the lat_proc and lat_ctx tests to measure the over-

Table 4: Energy Savings/Performance Delay Results

Bench.	Low α		Med α		High α	
	%delay	%energy	%delay	%energy	%delay	%energy
qsort	6	17.4	16.7	32	25	41
djpeg	7	21	15.2	37.2	26.5	45.2
dgzip	15	30	21	42	27.7	49
bf	6	11	16	27.6	25	40

(a) Single Tasks

Bench.	Low α		Med α		High α	
	%delay	%energy	%delay	%energy	%delay	%energy
qsort+djpeg	6	17.4	15.4	33	24.6	41
dgzip+djpeg	12.8	24	19	39.5	27	48.7
qsort+dgzip	7	19.7	17.7	34.6	25.7	42.3
dgzip+bf	11	18	20	33	26	45

(b) Multi-Tasking

head added to process creation and context switch times. The lat_ctx test is performed for context switches between 20 processes (the maximum it supports). We observe that the overhead is negligible in both the cases. For lat_proc it is almost 0%, while for lat_ctx it is around 3%. The overhead is negligible because the event processing functions in the LKM are extremely fast and lightweight. During task creation we just initialize the weights, which is a quick operation. The controller itself is implemented in fixed point arithmetic and is extremely lightweight. As discussed in section 4, the use of weights obviates the need of storing previous PMU samples thereby avoiding a potential overhead.

Comparison to prior work: Our work, in contrast to the previous approaches, represents the first full implementation of DVFS for a multitasking environment with negligible overhead and a simple mechanism for trading off energy with performance. The work done in [16] is a system level DVFS implementation that supports multi-tasking and is lightweight. However, their policy does not provide tunable e/p trade-off and is fixed to a performance delay of 10%. In [17] and [18], the techniques provide finely tuned control over energy savings and performance delay. They also achieve high energy savings due to the fine-grained v-f settings of their platform as discussed earlier in this section. Our performance delay is similar to the one they present in [17], since that platform, like ours, does not support external memory accesses events (MEM). In platform used in [18] the delay is lower since it supports MEM events. Clearly, MEM is a more accurate indicator of memory-intensiveness. However, neither [17] nor [18] have experiments in a multitasking environment. Their regression based approach involves higher overhead, since it has to maintain and operate on a queue of 25 most recent PMU samples. In their tests with lmbench, they increase the context switch time by a factor of 2 from 100μs to 200μs, while our overhead is negligible. Although that is small compared to the current scheduler quantum in Linux (10ms), some embedded operating systems, e.g. Windows CE 3.0 [24], have already reduced the scheduler quantum to 1ms. In such systems, such an increase in context switch times represents 10% of a constant overhead, a significant fraction.

6. CONCLUSION

In this paper we presented the design, the implementation, and the experimental evaluation of an extremely lightweight DVFS technique for a multi-tasking environment. The technique performs DVFS on the basis of accurate task characterization using runtime statistics provided by the platform and our online learning algorithm. The online learning algorithm guarantees fast convergence to the best setting for the current workload based on each task's characteristics. It achieves near maximum savings in both single task and multi-task environments at an overhead which is a factor of 2 less than the state of the art techniques.

7. REFERENCES

[1] "Intel XScale Core Developer's Manual," http://download.intel.com/design/intelxscale/27347302.pdf.

[2] "Mobile AMD Athlon 4 processor model 6 cpga data sheet," http://www.amd.com, 2001.

[3] M. Fleischmann, "LongRun Power Management - Dynamic Power Management for Crusoe Processors," http://www.transmeta.com/crusoe/longrun.html.

[4] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," In *FOCS*, 1995.

[5] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," In *ISLPED*, 1998.

[6] G. Quan and X. Hu, "Minimum energy fixed-priority scheduling for variable voltage processor," In *DATE*, 2002.

[7] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava, "Synthesis techniques for low-power hard real-time systems on variable voltage processors," In *RTSS*, 1998.

[8] D. Shin, J. Kim, and S. Lee, "Low-energy intra-task voltage scheduling using static timing analysis," In *DAC*, 2001.

[9] S. Lee and T. Sakurai, "Run-time power control scheme using software feedback loop for low-power real-time application," In *ASP-DAC*, 2000.

[10] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, "Profile-based dynamic voltage scheduling using program checkpoints," In *DATE*, 2002.

[11] C. H. Hsu and U. Kremer, "Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches," In *PACS*, 2002.

[12] E. Y. Chung, G. D. Micheli, and L. Benini, "Contents provider-assisted dynamic voltage scaling for low energy multimedia applications," In *ISLPED*, 2002.

[13] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins, "Energy-aware runtime scheduling for embedded-multiprocessor SOCs," *IEEE Design & Test of Computers*, 2001.

[14] D. Marculescu, "On the use of microarchitecture-driven dynamic voltage scaling," *Workshop on Complexity-Effective Design*, 2000.

[15] S. Ghiasi, J. Casmira, and D. Grunwald, "Using ipc variation in workloads with externally specified rates to reduce power consumption." *Workshop on Complexity-Effective Design*, 2000.

[16] A. Weissel and F. Bellosa, "Process cruise control: Event-driven clock scaling for dynamic power management," In *CASES*, 2002.

[17] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," In *ISLPED*, 2004.

[18] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *IEEE Trans. on CAD*, 2005.

[19] G. Dhiman and T. S. Rosing, "Dynamic power management using machine learning," In *ICCAD*, 2006.

[20] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," *SIGOPS Oper. Syst. Rev.*, 2006.

[21] Freund and Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *JCSS*, vol. 55, 1997.

[22] http://www.eecs.umich.edu/mibench/.

[23] http://www.bitmover.com/lmbench/.

[24] http://www.microsoft.com/msj/0799/wincekernel/wincekernel.aspx.